

## Modulo 1

**JS INSTITUTE**  
Open Education & Development Group

« 1.3.1.3 El Programa "¡Hola, Mundo!" - Entorno de desarrollo en línea »

**Sandbox**

**Entorno de desarrollo en línea**

Afortunadamente, nuestra plataforma utiliza un entorno en línea listo para usarse, como mencionamos en el capítulo anterior. El entorno OpenEDG te permite editar y ejecutar programas escritos en JavaScript. Toma en cuenta que la parte de la pantalla dedicada a este entorno se divide en tres partes. La parte superior es el editor, donde podemos elegir si editar un archivo JavaScript, HTML o CSS (diremos algunas palabras sobre HTML y CSS en un momento). Todos estos archivos juntos forman el código a ejecutar en nuestro entorno de entrenamiento. Nos interesarán principalmente la pestaña del archivo JavaScript: app.js. En la parte inferior izquierda de la pantalla, hay una ventana que simula la consola, en la que aparecerán los mensajes del intérprete y la información que escribimos. La ventana del lado derecho está diseñada para mostrar la página en cuyo contexto se ejecuta nuestro código JavaScript. Esta ventana será la menos útil en esta parte del curso.

En el editor, deberías ver la pieza de código que acabamos de discutir, que contiene la función console.log. Intenta ejecutarlo. Debes presionar el botón resultado con el ícono de reproducción, ubicado directamente sobre el editor. Como resultado, la ventana inferior que simula la consola debería mostrar:

```
¡Hola, Mundo!
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**JS INSTITUTE**  
Open Education & Development Group

« 1.3.1.10 Resumen de Sección - El Programa "¡Hola, Mundo!" »

**Sandbox**

**Resumen**

Nuestro primer programa se lanzó en un entorno en línea al principio. Este entorno nos permite ocultar ciertos detalles que no son importantes para nosotros en esta etapa del curso. Todos los ejercicios y ejemplos que discutiremos deben realizarse en este entorno.

Sin embargo, de vez en cuando, sería bueno que intentaras hacer el ejemplo elegido también en el entorno local. Está mucho más cerca de lo que realmente se usa en el trabajo de un desarrollador web. Ejecutar código JavaScript en el entorno local puede parecer un poco engorroso al principio, pero afortunadamente esta es solo una primera impresión. Recuerda, para probar instrucciones simples, solo necesitas usar la consola con una página vacía (por ejemplo, about:blank). Si deseas probar un código un poco más grande, es mejor crear un archivo html que se refiera al archivo que contiene nuestro código JavaScript usando la etiqueta `<script>`.

```
1
```

**app.js**

**Console >...**

```
¡Hola, Mundo!
```

**Fullscreen**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

The screenshot shows a browser-based development environment for OpenEDG. On the left, the code editor displays `index.html` with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title> Brayan Daniel</title>
</head>
<body>
  <h1>Brayan Daniel</h1>
  <!-- Enlaza el archivo JavaScript -->
  <script src="script.js"></script>
</body>
</html>
```

The right side shows a preview window titled "Brayan Daniel" displaying the rendered HTML with the heading "Brayan Daniel". Below the preview is a "Console" window showing the output: "¡Hola Brayan Daniel!".

This screenshot shows the same browser-based development environment for OpenEDG. Now, the code editor displays `app.js` with the following content:

```
// Código JavaScript de ejemplo.
console.log("¡Hola Brayan Daniel!");
```

The preview window still shows "Brayan Daniel" and the console output remains "¡Hola Brayan Daniel!".

The screenshot shows the JS INSTITUTE Tareas interface. At the top, it says "1.3.1.11 El Programa \"¡Hola, Mundo!\" - Tareas".

**Tarea 1**  
Emplea `console.log` para enviar tu nombre completo a la consola.  
**Ejemplo**

```
1 console.log("Brayan Mendiola");
```

**Tarea 2**  
Muestra tu año de nacimiento.  
**Ejemplo**

```
app.js
Console>_
Brayan Mendiola
```

**Tarea 3**  
Vuelve a intentar mostrar tu año de nacimiento, esta vez pasando la fecha sin las comillas.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Tarea 2**

Muestra tu año de nacimiento.

**Ejemplo**

**Tarea 3**

Vuelve a intentar mostrar tu año de nacimiento, esta vez pasando la fecha sin las comillas.

**Ejemplo**

**Tarea 4**

Podemos pasar varios argumentos a `console.log` separados por comas, por ejemplo:

```
1 console.log("2003");
```

app.js

Console >...

2003

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Ejemplo**

**Tarea 3**

Vuelve a intentar mostrar tu año de nacimiento, esta vez pasando la fecha sin las comillas.

**Ejemplo**

**Tarea 4**

Podemos pasar varios argumentos a `console.log` separados por comas, por ejemplo:

```
1 console.log("abc", "def", "ghi");
```

Envía información sobre ti a la consola en el formato: Nombre Apellido (Año) por ejemplo Mary Stuart (1542).

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Tarea 4**

Podemos pasar varios argumentos a `console.log` separados por comas, por ejemplo:

```
1 console.log("abc", "def", "ghi");
```

Envía información sobre ti a la consola en el formato: Nombre Apellido (Año) por ejemplo Mary Stuart (1542).

- Enviar toda la información como un solo argumento.
- Enviar la información nombre, apellido, año como argumentos separados.

**Ejemplo**

**Tarea 5**

Envía la misma información (nombre, apellido, año) a la consola, no una

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

« 1.3.1.11 El Programa "¡Hola, Mundo!" - Tareas »

Ejemplo

**Tarea 5**

Envía la misma información (nombre, apellido, año) a la consola, no una al lado de la otra, sino en líneas consecutivas.

Ejemplo

**Tarea 6**

Una cadena se puede concatenar usando el signo `+`, por ejemplo `"abc" + "def"` serán tratados como `"abcdef"`. Intenta escribir tu nombre, apellido y año de nacimiento en una línea nuevamente, esta vez no separados por comas, sino por el signo `+`.

Ejemplo

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

Console >... Fullscreen

```
app.js
Brayan
Mendiola
(2003)
```

**JS INSTITUTE**  
Open Education & Development Group

« 1.3.1.11 El Programa "¡Hola, Mundo!" - Tareas »

Ejemplo

al lado de la otra, sino en líneas consecutivas.

Ejemplo

**Tarea 6**

Una cadena se puede concatenar usando el signo `+`, por ejemplo `"abc" + "def"` serán tratados como `"abcdef"`. Intenta escribir tu nombre, apellido y año de nacimiento en una línea nuevamente, esta vez no separados por comas, sino por el signo `+`.

Ejemplo

**Tarea 7**

Coloca espacios en los lugares apropiados, de modo que cuando se muestre, se obtenga el mismo efecto que en la **Tarea 4**.

Ejemplo

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

Console >... Fullscreen

```
app.js
Brayan Mendiola(2003)
```

**JS INSTITUTE**  
Open Education & Development Group

« 1.3.1.11 El Programa "¡Hola, Mundo!" - Tareas »

**Tarea 6**

Una cadena se puede concatenar usando el signo `+`, por ejemplo `"abc" + "def"` serán tratados como `"abcdef"`. Intenta escribir tu nombre, apellido y año de nacimiento en una línea nuevamente, esta vez no separados por comas, sino por el signo `+`.

Ejemplo

**Tarea 7**

Coloca espacios en los lugares apropiados, de modo que cuando se muestre, se obtenga el mismo efecto que en la **Tarea 4**.

Ejemplo

```
console.log("Mary " + " Stuart " + "(1542);
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

Console >... Fullscreen

```
app.js
Brayan Mendiola (2003)
```

## Modulo 2

### Sección 1

**JS INSTITUTE**  
Open Education & Development Group

« 2.1.1.4 Variables - Declarando variables »

**Declarando variables**

Como mencionamos anteriormente, **declaramos** la variable para reservarle un nombre. Esto es una simplificación, porque de hecho, el espacio de memoria también está reservado para la variable, pero al programar en JavaScript, prácticamente nunca tenemos que pensar en lo que sucede en la memoria. Por lo general, los valores almacenados en la variable podrán modificarse durante la ejecución del programa (después de todo, son "variables"). ¿Por qué? Porque podemos declarar variables cuyos valores no se pueden cambiar. Para ser honesto, ya ni siquiera las llamamos variables, las llamamos **constantes**. Para las declaraciones, usamos las palabras clave `var` o `let` para **variables** y `const` para **constantes**. Por ahora, sin embargo, sigamos con las variables habituales y volveremos a las constantes en un momento.

Analicemos el siguiente ejemplo de código (también lo encontrarás en la ventana del editor; ejecútalo allí y observa los resultados en la consola):

```
var height;
console.log(height); // -> undefined
console.log(weight); // -> Uncaught ReferenceError: weight is not defined
```

app.js

Console >...

undefined  
Uncaught ReferenceError: weight is not defined

X Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**JS INSTITUTE**  
Open Education & Development Group

« 2.1.1.4 Variables - Declarando variables »

**Declarando variables**

Como mencionamos anteriormente, **declaramos** la variable para reservarle un nombre. Esto es una simplificación, porque de hecho, el espacio de memoria también está reservado para la variable, pero al programar en JavaScript, prácticamente nunca tenemos que pensar en lo que sucede en la memoria. Por lo general, los valores almacenados en la variable podrán modificarse durante la ejecución del programa (después de todo, son "variables"). ¿Por qué? Porque podemos declarar variables cuyos valores no se pueden cambiar. Para ser honesto, ya ni siquiera las llamamos variables, las llamamos **constantes**. Para las declaraciones, usamos las palabras clave `var` o `let` para **variables** y `const` para **constantes**. Por ahora, sin embargo, sigamos con las variables habituales y volveremos a las constantes en un momento.

Analicemos el siguiente ejemplo de código (también lo encontrarás en la ventana del editor; ejecútalo allí y observa los resultados en la consola):

```
var height;
console.log(height); // -> undefined
console.log(weight); // -> Uncaught ReferenceError: weight is not defined
```

app.js

Console >...

undefined

X Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**Sandbox**

## Variables - Declarando variables cont.

En el ejemplo, usamos la palabra clave `var`. La alternativa es la palabra clave `let`. Usamos ambas palabras clave de la misma manera. Ambas están destinados para declarar variables, y ambas se pueden encontrar en diferentes ejemplos en Internet o en libros. Sin embargo, no son exactamente iguales y discutiremos las diferencias en su funcionamiento más adelante en este capítulo (incluso en varios lugares).

La palabra clave `var` proviene de la sintaxis JavaScript original, y la palabra clave `let` se introdujo mucho más tarde. Por lo tanto, encontrarás var en programas más antiguos. Actualmente, se recomienda enfáticamente usar la palabra `let` por razones que discutiremos en un momento.

Entonces, echemos un vistazo a nuestro ejemplo reescrito esta vez usando la palabra clave `let`.

```
let height;
console.log(height); // -> undefined
```

The screenshot shows a browser-based JavaScript environment. At the top, there's a toolbar with icons for file operations and a 'Sandbox' button. Below the toolbar is a code editor window containing the following code:

```
1 var height;
2 console.log(height); // -> undefined
3 console.log(weight); // -> Uncaught ReferenceError: weight is not defined
```

Below the code editor is a 'Console' window with the output:

```
app.js
Console >...
undefined
Uncaught ReferenceError: weight is not defined
```

At the bottom of the interface, there's a message: "We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X".

**Sandbox**

## Inicializando variables

Después de una declaración exitosa, la variable debe ser **inicializada**, en otras palabras, debe recibir su primer valor. La **inicialización** se realiza asignando un determinado valor a una variable (indicado por su nombre). Para asignarlo usamos el operador `=`.

```
let height = 180;
let anotherHeight = height;
```

Puedes asignar a una variable: un valor específico; el contenido de otra variable; o, por ejemplo, el resultado devuelto por una función.

La inicialización se puede realizar junto con la declaración o por separado como un comando independiente. Es importante ingresar el primer valor en la variable antes de intentar leerla, modificarla o mostrarla.

```
let height = 180;
let anotherHeight = height;
let weight;
console.log(height); // -> 180
```

The screenshot shows a browser-based JavaScript environment. At the top, there's a toolbar with icons for file operations and a 'Sandbox' button. Below the toolbar is a code editor window containing the following code:

```
1 let height = 180;
2 console.log(height); // -> 180
3 console.log("height"); // -> height
4
5
```

Below the code editor is a 'Console' window with the output:

```
app.js
Console >...
180
height
```

At the bottom of the interface, there's a message: "We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X".

**Sandbox**

## Declaraciones y modo estricto

JavaScript tuvo algunos cambios importantes introducidos en 2009 y 2015. La mayoría de estos cambios ampliaron la sintaxis del lenguaje con nuevos elementos, pero algunos de ellos se referían solo al funcionamiento de los intérpretes de JavaScript. A menudo se trataba de aclarar el comportamiento de los intérpretes en situaciones potencialmente erróneas, como en los casos de inicialización de variables sin ninguna declaración previa.

Veamos un ejemplo:

```
height = 180;
console.log(height); // -> 180
```

The screenshot shows a browser-based JavaScript environment. At the top, there's a toolbar with icons for file operations and a 'Sandbox' button. Below the toolbar is a code editor window containing the following code:

```
1 height = 180;
2 console.log(height); // -> 180
3
4
5
```

Below the code editor is a 'Console' window with the output:

```
app.js
Console >...
180
```

At the bottom of the interface, there's a message: "We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X".

## Constantes

La palabra clave `const` se usa para declarar contenedores similares a variables. Dichos contenedores se denominan **constantes**. Las constantes también se utilizan para almacenar ciertos valores, pero una vez que se han ingresado valores durante la inicialización, ya no se pueden modificar. Esto significa que este tipo de contenedor se declara e inicializa simultáneamente. Por ejemplo, la siguiente declaración de la constante greeting es correcta:

```
const greeting = "¡Hola!";
```

Pero esta próxima definitivamente causa un error:

```
const greeting; // -> Uncaught SyntaxError: Missing initializer for variable 'greeting'.
greeting = "¡Hola!";
```

Como dijimos, un cambio en la constante es imposible. Esta vez la declaración es correcta, pero tratamos de modificar el valor almacenado en la constante.

```
1 const greeting = "¡Hola!";
2 console.log(greeting);
3
```

app.js

Console>...

¡Hola!

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Alcance

Hasta ahora, asumimos que después de declarar una variable, su nombre podría usarse en todo el código del programa (es decir, el alcance de la variable es global). Esto no es del todo cierto: el alcance de una variable depende de dónde se declare. Desafortunadamente, para una buena comprensión del alcance de una variable, necesitamos aprender algunos elementos de programación más, como instrucciones o funciones condicionales, que se analizarán con detalle más adelante en el curso. Así que aquí nos limitaremos a la información básica y volveremos a este tema en diferentes partes del curso. Uno de los elementos básicos que influyen en el alcance de las variables es un **bloque del programa**.

### Bloques del Programa

Podemos separar el código de un programa en bloques. En los bloques que creamos usando llaves {}, hay un conjunto de instrucciones que, por alguna razón, deben tratarse de forma independiente. Los bloques suelen estar asociados a instrucciones condicionales, bucles o funciones, de las que hablaremos más adelante. También podemos separar un bloque de un programa que no tenga nada en especial, simplemente eligiendo un determinado rango de instrucciones (en la práctica, esto no está

```
1 let counter;
2 console.log(counter); // -> undefined
3 +
4 {
5   counter = 1;
6   console.log(counter); // -> 1
7 }
8
9 counter = counter + 1;
10 console.log(counter); // -> 2
11
12
13
```

app.js

Console>...

undefined

1

2

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

Es hora de seguir adelante para determinar qué está pasando realmente con estos alcances. Desafortunadamente, los alcances de las variables (y constantes) declaradas con `let` y `const` son ligeramente diferentes a las declaradas con `var`. Así que los discutiremos de forma independiente.

```
let y | const
```

La primera regla es simple. Si declaramos alguna variable o constante usando `let` o `const`, respectivamente, fuera de los bloques de código, serán **globales**. Con esto queremos decir que sus nombres serán visibles en todo el programa, fuera de los bloques, dentro de los bloques, en las funciones, etc. Podremos referirnos a ellos en cualquier lugar por sus nombres y, por supuesto, tendremos acceso a sus valores.

¿Qué sucede si declaramos algo usando `let` o `const` dentro de un bloque? Esto creará una variable o constante local. Será visible solo dentro del bloque en el que se declaró y en los bloques que opcionalmente se pueden anidar en él.

Veamos un ejemplo sencillo:

```
let height = 180;
{
```

```
1 let height = 200;
2 +
3   let weight = 100;
4   {
5     let info = "tall";
6     console.log(height); // -> 200
7     console.log(weight); // -> 100
8     console.log(info); // -> tall
9   }
10
11
12
13
14
15
16
```

app.js

Console>...

200

100

tall

200

100

Uncaught ReferenceError: info is not defined

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

[Sandbox](#)

```
var
```

En el caso de declaraciones de variables usando la palabra clave `var`, la situación es ligeramente diferente. La variable declarada utilizándola fuera de los bloques será, como en el caso de `let`, `global`, es decir, será visible en todas partes. Si la declaras dentro de un bloque, entonces... bueno, por lo general volverá a ser `global`.

Comencemos con un ejemplo simple:

```
var height = 180;
{
  var weight = 70;
  console.log(height); // -> 180
  console.log(weight); // -> 70
}
console.log(height); // -> 180
console.log(weight); // -> 70
```

Como era de esperar, ambas variables, `height` y `weight`, resultan ser

```
1 var height = 180;
2 {
3   var weight = 70;
4   console.log(height); // -> 180
5   console.log(weight); // -> 70
6 }
7 console.log(height); // -> 180
8 console.log(weight); // -> 70
9
10
11
```

app.js

Console>...

180

70

180

70

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

[Sandbox](#)

La forma de definir la función que se muestra en el ejemplo es una de varias disponibles en JavaScript. La definición comienza con la palabra clave `function`, seguida del nombre de la función que inventamos. Después del nombre, verás paréntesis, que opcionalmente podrían contener parámetros pasados a la función (volveremos a esto cuando analicemos funciones con más precisión). Luego abrimos el bloque del programa, que contiene las instrucciones pertenecientes a la función. Al definir una función, las instrucciones contenidas en la función no se ejecutan. Para ejecutar la función, debes llamarla de forma independiente, usando su nombre.

Echa un vistazo al siguiente programa.

```
console.log("Comencemos!"); // -> Comencemos;
console.log("Hola"); // -> Hola
console.log("Mundo"); // -> Mundo
console.log("y otra vez"); // -> y otra vez:
console.log("Hola"); // -> Hola
console.log("Mundo"); // -> Mundo
console.log("y una vez más"); // -> y una vez más:
console.log("Hola"); // -> Hola
console.log("Mundo"); // -> Mundo
```

```
1 console.log("Comencemos!"); // -> Comencemos;
2 console.log("Hola"); // -> Hola
3 console.log("Mundo"); // -> Mundo
4 console.log("y otra vez"); // -> y otra vez:
5 console.log("Hola"); // -> Hola
6 console.log("Mundo"); // -> Mundo
7 console.log("y una vez más"); // -> y una vez más:
8 console.log("Hola"); // -> Hola
9 console.log("Mundo"); // -> Mundo
10
11
12
```

app.js

Console>...

Comencemos:

Hola

Mundo

y otra vez:

Hola

Mundo

y una vez más:

Hola

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

[Sandbox](#)

### La palabra clave `var` - continuación

Después de esta breve introducción a las funciones (obviamente, esta no es nuestra última reunión con ellas), volvamos a la palabra clave `var` y los ámbitos de las variables.

Si declaramos una variable usando la palabra clave `var` dentro de una función, su alcance se limitará solo al interior de esa función (es un alcance local). Esto significa que el nombre de la variable se reconocerá correctamente solo dentro de esta función.

Consideremos el siguiente ejemplo:

```
var globalGreeting = "Buenos";
function testFunction() {
  var localGreeting = "Días";
  console.log("función:");
  console.log(globalGreeting);
  console.log(localGreeting);
}
```

```
1 var globalGreeting = "Buenos";
2
3 function testFunction() {
4   var localGreeting = "Días";
5   console.log("función:");
6   console.log(globalGreeting);
7   console.log(localGreeting);
8 }
9
10 testFunction();
11
12 console.log("programa principal:");
13 console.log(globalGreeting);
14 console.log(localGreeting); // -> Uncaught ReferenceError: localGreeting is not defined
15
16
```

app.js

Console>...

función:

Buenos

Días

programa principal:

Buenos

Uncaught ReferenceError: localGreeting is not defined

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

## Sombreado

JavaScript permite sombreado de variables. ¿Qué significa eso? Significa que podemos declarar una variable global y una variable local con el mismo nombre.

En el alcance local, en el que declaramos una variable local usando su nombre, tendremos acceso al valor local (la variable global está oculta detrás de la local, por lo que no tenemos acceso a ella en este alcance local). Usar este nombre fuera del alcance local significa que nos referiremos a la variable global. **Sin embargo, esta no es la mejor práctica de programación y debemos evitar tales situaciones.** No es difícil advinar que con un poco de falta de atención, el uso de este mecanismo puede conducir a situaciones no deseadas y probablemente a errores en el funcionamiento del programa.

Si queremos evitar tales situaciones, sería bueno ver exactamente de qué se trata. Comencemos con un ejemplo sin sombreado:

```
let counter = 100;
console.log(counter); // -> 100
{
  counter = 200;
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 app.js

 Console...

 Fullscreen

100

200

100

 Sandbox

## Sombreado - continuación

El sombreado no solo puede estar relacionado con la situación en la que una variable local cubre una variable global. Si aparecen alcances anidados (por ejemplo, bloques anidados en el caso de una declaración let), la variable local declarada en un bloque más anidado eclipsará la variable local del mismo nombre declarada en el bloque externo.

El sombreado también está presente en las declaraciones de variables que usan la palabra `var`, y esta vez el alcance local no está limitado por el bloque de programa, sino por el bloque de funciones.

```
var counter = 100;

function testFunction() {
  var counter = 200;
  console.log(counter);
}

console.log(counter); // -> 100
testFunction(); // -> 200
console.log(counter); // -> 100
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 app.js

 Console...

 Fullscreen

100

200

100

 Sandbox

## Hoisting

¿Recuerdas que dijimos que todas las variables deben declararse antes de su uso? Esto no es del todo cierto, y realmente la palabra "debería" encaja mejor que "debe". Por supuesto, una buena práctica es siempre declarar las variables antes de usarlas. Y apégate a esto. Pero la sintaxis de JavaScript original permite algunas desviaciones de esta regla.

El intérprete de JavaScript escanea el programa antes de ejecutarlo, buscando errores en su sintaxis, entre otras cosas. Hace una cosa más en esta ocasión. Busca todas las declaraciones de variables y las mueve al principio del rango en el que fueron declaradas (al principio del programa si son globales, al principio del bloque si es una declaración let local, o al principio de la función si es una declaración local var). Todo esto sucede, por supuesto, en la memoria del intérprete, y los cambios no son visibles en el código.

**Hoisting**, es un mecanismo bastante complejo y francamente bastante incoherente. Comprenderlo bien requiere la capacidad de usar libremente muchos elementos de JavaScript, que aún no hemos mencionado.

Además, es más una curiosidad que algo práctico que usarás a escribir programas, por lo que veremos solo un pequeño ejemplo que nos permitirá

 app.js

 Console...

 Fullscreen

180

undefined

70

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## &lt;&gt; 2.1.1.19 Variables - Tareas &lt;&gt;

**Tareas****Tarea 1**

Juguemos a la floristería. Declara seis variables, recordando nombrarlas según su propósito:

- el precio de una sola rosa (8) y el número de rosas que tienes (70)
- el precio de un solo lirio (10) y el número de lirios que tienes (50)
- el precio de un solo tulipán (2) y la cantidad de tulipanes que tienes (120)

Ahora declara tres variables, una para cada una de las rosas, lirios y tulipanes que tienes, en las que colocas su precio total. Inserta los valores correspondientes en las variables utilizando las variables declaradas en el paso anterior. Finalmente, declara una variable en la que almacenes el precio de todas tus flores (nuevamente, usa las variables anteriores para la inicialización). Muestra toda la información del inventario en la consola de la siguiente forma:

```
Rosa: precio unitario: 8 , cantidad: 70 , valor: 560
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

app.js

Console >...

```
Rosa: precio unitario: 8 , cantidad: 70 , valor: 560
Lirio: precio unitario: 10 , cantidad: 50 , valor: 500
Tulipán: precio unitario: 2 , cantidad: 120 , valor: 240
Total: 1300
```

Fullscreen

## &lt;&gt; 2.1.1.19 Variables - Tareas &lt;&gt;

**Solución****Tarea 2**

Modifica el código del ejemplo anterior. Supón que los precios de las flores serán constantes (no cambiarán). Declara e inicializa las variables restantes de la misma manera que en el ejemplo anterior. Muestra toda la información recopilada en la consola. Ahora disminuye el número de rosas en 20 y el de lirios en 30. Vuelve a mostrar toda la información recopilada en la consola.

Solución

## &lt;&gt; 2.1.1.20 LABORATORIO: Variables &lt;&gt;

**LABORATORIO****Tiempo Estimado**

15-30 minutos

**Nivel de Dificultad**

Fácil

**Objetivos**

Familiarizar al estudiante con:

- variables (es decir, nombrar, declarar, inicializar y modificar su valor)

**Escenario**

Nuestra tarea será crear una lista de contactos. Inicialmente, la lista será bastante simple: solo escribirímos tres personas utilizando los datos que se muestran en la tabla a continuación. En el resto del curso, volverá a este script y lo ampliará sistemáticamente con nuevas funciones, utilizando los elementos de JavaScript recién aprendidos.

app.js index.html style.css

Console >...

Información del primer contacto:  
Nombre: Maxwell Wright  
Teléfono: (0191) 719 6495

Correo: Curabitur.egestas.nunc@nonummyac.co.uk

Información del último contacto:

Nombre: Helen Richards

Teléfono: 0800 1111

Correo: libero@convallis.edu

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Modulo 2

### Sección 2

**JS INSTITUTE**  
Open Education & Development Group

« 2.2.1.3 Tipos de Datos - El operador typeof »

**El operador `typeof`**

Mientras aprendes sobre los tipos de datos de JavaScript, el operador `typeof` puede ser útil. De hecho, también es útil para el trabajo normal con este lenguaje, por lo que sería bueno que lo recordaras para más adelante. Uno de los capítulos posteriores lo dedicaremos a los **operadores**, pero en este punto es suficiente saber que un operador es un símbolo o nombre que representa alguna acción a realizar sobre los argumentos indicados. Por ejemplo, el símbolo `+` es un operador de dos argumentos que representa la suma.

El operador `typeof` que acabamos de mencionar es unario (solo toma un argumento) y nos informa del tipo de datos indicados como un argumento dado. El argumento puede ser un literal o una variable; en este último caso, se nos informará sobre el tipo de datos almacenados en él. El operador `typeof` devuelve una cadena con uno de los valores fijos asignados a cada uno de los tipos.

Todos los posibles valores de retorno del operador `typeof` son:

```
"undefined"  
"object"
```

app.js

```
let year = 1990;  
console.log(typeof year); // -> number  
console.log(typeof 1991); // -> number  
  
let name = "Alice";  
console.log(typeof name); // -> string  
console.log(typeof "Bob"); // -> string  
  
let typeOfYear = typeof year;  
console.log(typeOfYear); // -> number  
console.log(typeof typeOfYear); // -> string  
  
13  
14
```

Console>...  
number  
number  
string  
string  
number  
string

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policy](#). X

**JS INSTITUTE**  
Open Education & Development Group

« 2.2.1.4 Tipos de Datos Primitivos » Boolean (booleano)

**Tipos de Datos Primitivos**

En JavaScript, hay seis tipos de datos primitivos (o simples): **Boolean**, **Number**, **BigInt**, **String**, **Symbol** y **undefined**. Además, el valor primitivo **null** también se trata como un tipo separado. Un dato primitivo, como ya hemos dicho, es un tipo de dato cuyos valores son atómicos. Esto significa que el valor es un elemento indivisible.

Intentemos echar un vistazo más de cerca a los datos primitivos.

**Boolean**

El Boolean (booleano) es un tipo de dato lógico. Solo puede tomar uno de dos valores: `true` o `false`. Se usa principalmente como una expresión condicional necesaria para decidir qué parte del código debe ejecutarse o cuánto tiempo debe repetirse algo (esto se denomina declaración de flujo de control y lo veremos más de cerca en el Módulo 4).

Los valores booleanos también se utilizan como lo que comúnmente se conoce como una **bandera**, una variable que señala algo que puede estar presente o ausente, habilitado o deshabilitado, etc. Como cualquier otra variable, los valores booleanos deben tener nombres claros e informativos.

```
let isDataValid = true;  
let isStringTooLong = false;  
let isGameOver = false;  
continueLoop = true;  
  
6 console.log(false); // -> false  
7 console.log(typeof false); // -> boolean  
8 console.log(isDataValid); // -> true  
9 console.log(typeof isDataValid); // -> boolean  
10  
11  
12
```

app.js

```
false  
boolean  
true  
boolean
```

Console>...  
false  
boolean  
true  
boolean

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policy](#). X

**Number (numérico)**

Este es el tipo numérico principal en JavaScript que representa tanto números reales (por ejemplo, fracciones) como enteros. El formato en el que se almacenan los datos de este tipo en la memoria hace que los valores de este tipo sean a veces aproximados (especialmente, pero no únicamente, valores muy grandes o algunas fracciones). Se supone, entre otras cosas, que para garantizar la exactitud de los cálculos, los valores enteros deben limitarse en JavaScript al rango de  $-(2^{53} - 1)$  a  $(2^{53} - 1)$ .

**Los números** permiten todas las operaciones aritméticas típicas, como suma, resta, multiplicación y división.

```
const year = 1991;
let delayInSeconds = 0.00016;
let area = (16 * 3.14);
let halfArea = area / 2;

console.log(year); // -> 1991
console.log(typeof year); // -> number;
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy and cookie policies](#). X

numeros reales (por ejemplo, fracciones) como enteros. El formato en el que se almacenan los datos de este tipo en la memoria hace que los valores de este tipo sean a veces aproximados (especialmente, pero no únicamente, valores muy grandes o algunas fracciones). Se supone, entre otras cosas, que para garantizar la exactitud de los cálculos, los valores enteros deben limitarse en JavaScript al rango de  $-(2^{53} - 1)$  a  $(2^{53} - 1)$ .

**Los números** permiten todas las operaciones aritméticas típicas, como suma, resta, multiplicación y división.

```
const year = 1991;
let delayInSeconds = 0.00016;
let area = (16 * 3.14);
let halfArea = area / 2;

console.log(year); // -> 1991
console.log(typeof year); // -> number;
```

Los números en JavaScript generalmente se presentan en forma decimal, a la que estamos acostumbrados en la vida cotidiana. Sin embargo, si un literal

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy and cookie policies](#). X

**BigInt**

El tipo de dato **BigInt** no se usa con demasiada frecuencia. Nos permite escribir números enteros de prácticamente cualquier longitud. Para casi todas las operaciones numéricas normales, el tipo **Number** es suficiente, pero de vez en cuando necesitamos un tipo que pueda manejar números enteros mucho más grandes.

Podemos usar operaciones matemáticas en Bigints de la misma manera que en Numbers, pero hay una diferencia al dividir. Como BigInt es un tipo entero, el resultado de la división siempre se **redondeará hacia abajo** al número entero más cercano.

Los literales BigInt son números con el sufijo `bn`.

```
let big = 1234567890000000000000n;
let big2 = 1n;

console.log(big); // -> 1234567890000000000000n
console.log(typeof big); // -> bigint

console.log(big2); // -> 1n
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy and cookie policies](#). X

## String (cadenas)

El tipo String representa una secuencia de caracteres que forman un fragmento de texto. Las operaciones comunes en los textos incluyen la concatenación, la extracción de la subcadena y la verificación de la longitud de la cadena. Las cadenas se utilizan mucho en la programación y más aún en el desarrollo web, ya que tanto HTML como gran parte del contenido de Internet es texto.

El uso más común de texto en el desarrollo web incluye:

- Enlaces y rutas a recursos.
- Tokens.
- Comprobación de formularios y entradas llenadas por el usuario.
- Generación de contenido dinámico.

Las **Cadenas**, como otros datos primitivos, son inmutables, por lo que cuando queremos cambiar incluso una letra en una cadena, en realidad, creamos una nueva cadena.

En ejemplos anteriores, ya usamos cadenas de caracteres. Usamos comillas para indicar que un texto determinado debe tratarse como una cadena (es

```

1 let message1 = 'El buque \'Mars\' hizo escala en el puerto.';
2 let message2 = "El ciclón \"Cilida\" pasará cerca de Mauritius."
3
4 console.log(message1); // -> El buque 'Mars' hizo escala en el puerto.
5 console.log(message2); // -> El ciclón "Cilida" pasará cerca de Mauritius.
6
7 let path = "C:\\Windows";
8 console.log(path); // -> C:\\Windows
9
10
11

```

app.js

Console >...

El buque 'Mars' hizo escala en el puerto.  
El ciclón "Cilida" pasará cerca de Mauritius.  
C:\\Windows

Sandbox

Fullscreen

## String (cadenas)

Intentar realizar operaciones aritméticas en valores de tipo String, como resta, multiplicación o división, por lo general terminará en un error. Más precisamente, el valor de NaN se devolverá como resultado de la acción.

¿Por qué sucede esto? Al ver los operadores aritméticos `-`, `*` o `\`, el intérprete de JavaScript intenta interpretar los valores dados como números, o convertirlos en números. Entonces, si las cadenas de caracteres consisten en dígitos, la conversión automática será exitosa y obtendremos el resultado de la acción aritmética como un valor de tipo Number. Si la cadena de caracteres no puede interpretarse como un número (y convertirse), obtendremos el resultado NaN. Hablaremos más sobre las conversiones en un momento.

```

let path = "C:\\Windows" - "Windows";
console.log(path); // -> NaN

let test = "100" - "10";
console.log(test); // -> 90
console.log(typeof test); // -> number

```

```

1 let country = "Europa";
2 let continent = "España";
3
4 let sentence = `${country} se ubica en ${continent}.`;
5 console.log(sentence); // -> Malawi se ubica en Africa.
6
7
8

```

app.js

Console >...

Europa se ubica en España.

Sandbox

Fullscreen

## String (cadenas)

Intentar realizar operaciones aritméticas en valores de tipo String, como resta, multiplicación o división, por lo general terminará en un error. Más precisamente, el valor de NaN se devolverá como resultado de la acción.

¿Por qué sucede esto? Al ver los operadores aritméticos `-`, `*` o `\`, el intérprete de JavaScript intenta interpretar los valores dados como números, o convertirlos en números. Entonces, si las cadenas de caracteres consisten en dígitos, la conversión automática será exitosa y obtendremos el resultado de la acción aritmética como un valor de tipo Number. Si la cadena de caracteres no puede interpretarse como un número (y convertirse), obtendremos el resultado NaN. Hablaremos más sobre las conversiones en un momento.

```

let path = "C:\\Windows" - "Windows";
console.log(path); // -> NaN

let test = "100" - "10";
console.log(test); // -> 90
console.log(typeof test); // -> number

```

```

1 let country = "Malawi";
2 let continent = "Africa";
3
4 let sentence = `${country} se ubica en ${continent}.`;
5 console.log(sentence); // -> Malawi se ubica en Africa.
6
7
8
9

```

app.js

Console >...

Malawi se ubica en Africa.

Sandbox

Fullscreen

Intentar realizar operaciones aritméticas en valores de tipo String, como resta, multiplicación o división, por lo general terminará en un error. Más precisamente, el valor de NaN se devolverá como resultado de la acción.

¿Por qué sucede esto? Al ver los operadores aritméticos `-`, `*` o `\`, el intérprete de JavaScript intenta interpretar los valores dados como números, o convertirlos en números. Entonces, si las cadenas de caracteres consisten en dígitos, la conversión automática será exitosa y obtendremos el resultado de la acción aritmética como un valor de tipo Number. Si la cadena de caracteres no puede interpretarse como un número (y convertirse), obtendremos el resultado NaN. Hablaremos más sobre las conversiones en un momento.

```
let path = "C:\\Windows" - "Windows";
console.log(path); // -> NaN

let test = "100" - "10";
console.log(test); // -> 90
console.log(typeof test); // -> number
```

```
5 let resta = stringValue - 5; // Resta
6 let multiplicacion = stringValue * 2; // Multiplicación
7 let division = stringValue / 4; // División
8
9 console.log(suma); // -> "2010"
10 console.log(typeof suma); // -> string
11
12 console.log(resta); // -> 15
13 console.log(typeof resta); // -> number
14
15 console.log(multiplicacion); // -> 40
16 console.log(typeof multiplicacion); // -> number
17
18 console.log(division); // -> 5
19 console.log(typeof division); // -> number
20
```

app.js

Console >...

```
2010
string
15
number
40
number
5
```

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

Puedes hacer mucho trabajo útil con datos del tipo String. Desafortunadamente, requieren dos nuevos conceptos: **métodos** (e indirectamente, objetos) y **autoboxing**. La explicación exacta de ambos conceptos va más allá del alcance de este curso, por lo que intentaremos simplificarlos un poco.

En uno de los capítulos anteriores, presentamos el concepto de función, también en una forma algo simplificada. Ahora hablemos de métodos.

Un **método** es un tipo especial de función que pertenece a un objeto. Los **objetos** son tipos de datos complejos, que pueden constar de muchos valores (almacenados en propiedades) y métodos. Si deseas llamar al método de un objeto, escribe el nombre del método después de un punto. ¿Esto te recuerda a algo? Esta es exactamente la notación que usas cuando llamas a `console.log`. El objeto `consola` tiene muchos otros métodos además del método `log`, como `time` y `timeEnd` (que se pueden usar para medir el tiempo).

```
console.time();
console.log("probar consola"); // -> probar consola
console.timeEnd(); // -> default: 0.108154296875 ms
```

```
1 console.time();
2 console.log("probar consola"); // -> probar consola
3 console.timeEnd(); // -> default: 0.108154296875 ms
4
5
6
```

app.js

Console >...

```
undefined
probar consola
default: 0.299999998137355 ms
undefined
```

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

Los métodos de cadena y las propiedades comúnmente utilizados (es decir, valores con nombre relacionados con el objeto) son:

- `length`: propiedad que devuelve el número de caracteres en una cadena.

```
str = "java script language";
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
str.length -> 20
```

```
1 let str = "java script language";
2
3 console.log(str.length); // -> 20
4 console.log("test".length); // -> 4
5
6 console.log(str.charAt(0)); // -> 'j'
7 console.log('abc'.charAt(1)); // -> 'b'
8
9 console.log(str.slice(0, 4)); // -> 'java'
10 console.log("test".slice(1, 3)); // -> 'es'
11
12 console.log(str.split(' ')); // -> ['java', 'script', 'language']
13 ====== 14 ====== 15 ====== 16 ====== 17 ====== 18 ====== 19 ======
```

app.js

Console >...

```
20
4
j
b
java
es
java.script.language
192.168.1.1
```

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Undefined

El tipo undefined (Indefinido) tiene un solo valor: `undefined`. Es el valor predeterminado que tienen todas las variables después de una declaración si no se les asigna ningún valor. También puedes asignar el valor `undefined` a cualquier variable, pero en general, esto debe evitarse, porque si necesitamos marcar una variable para que no tenga ningún valor significativo, debemos usar `null`.

```
let declaredVar;
console.log(typeof declaredVar); // -> undefined

declaredVar = 5;
console.log(typeof declaredVar); // -> number

declaredVar = undefined;
console.log(typeof declaredVar); // -> undefined

//El operador typeof también puede devolver el valor indefinido
console.log(typeof notDeclaredVar); // -> undefined
```

```
2 console.log(typeof declaredVar); // -> undefined
3
4 declaredVar = 5;
5 console.log(typeof declaredVar); // -> number
6
7 declaredVar = undefined;
8 console.log(typeof declaredVar); // -> undefined
9
10 //El operador typeof también puede devolver el valor indefinido cuando se usa una variable
11
12
13 console.log(typeof notDeclaredVar); // -> undefined
14 console.log(notDeclaredVar); // -> Uncaught ReferenceError: notDeclared is not defined
15
16
17
```

**app.js**

**Console>**

undefined  
number  
undefined  
undefined  
Uncaught ReferenceError: notDeclaredVar is not defined




We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Undefined

El tipo undefined (Indefinido) tiene un solo valor: `undefined`. Es el valor predeterminado que tienen todas las variables después de una declaración si no se les asigna ningún valor. También puedes asignar el valor `undefined` a cualquier variable, pero en general, esto debe evitarse, porque si necesitamos marcar una variable para que no tenga ningún valor significativo, debemos usar `null`.

```
let declaredVar;
console.log(typeof declaredVar); // -> undefined

declaredVar = 5;
console.log(typeof declaredVar); // -> number

declaredVar = undefined;
console.log(typeof declaredVar); // -> undefined

//El operador typeof también puede devolver el valor indefinido
console.log(typeof notDeclaredVar); // -> undefined
```

```
1 let declaredVar;
2 console.log(typeof declaredVar); // -> undefined
3
4 declaredVar = 5;
5 console.log(typeof declaredVar); // -> number
6
7 declaredVar = undefined;
8 console.log(typeof declaredVar); // -> undefined
9
10 //El operador typeof también puede devolver el valor indefinido cuando se usa una variable
11
12
13 console.log(typeof notDeclaredVar); // -> undefined
14
15
16
```

**app.js**

**Console>**

undefined  
number  
undefined  
undefined

## null

El valor `null` es bastante específico. El valor en sí es primitivo, mientras que el tipo al que pertenece no es un tipo primitivo, como Number o undefined. Esta es una categoría separada, asociada con tipos complejos, como objetos. El valor `null` se usa para indicar que la variable no contiene nada y, en la mayoría de los casos, es una variable que pretende contener valores de tipos complejos.

En pocas palabras, podemos suponer que el valor `undefined` se asigna automáticamente a las variables no inicializadas, pero si queremos indicar explícitamente que la variable no contiene nada, le asignamos un valor `null`. Una advertencia importante para `null` es que cuando se verifica con el operador `typeof`, devolverá "`object`", un resultado sorprendente. Esto es parte de un sistema de objetos mucho más complicado, pero por ahora, solo necesitas saber que `typeof null` es igual a "`object`".

```
let someResource;
console.log(someResource); // -> undefined
console.log(typeof someResource); // -> undefined
```

```
1 let someResource;
2 console.log(someResource); // -> undefined
3 console.log(typeof someResource); // -> undefined
4
5 someResource = null;
6 console.log(someResource); // -> null
7 console.log(typeof someResource); // -> object
8
9
10
```

**app.js**

**Console>**

undefined  
undefined  
object




We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Conversiones de Tipo de Datos

### Funciones de construcción primitivas

Usar literales no es la única manera de crear variables de los tipos primitivos dados. La segunda opción es crearlos usando funciones del tipo **constructor**. Este tipo de funciones se utilizan principalmente en JavaScript para la programación orientada a objetos, que está fuera del alcance de nuestro curso. Sin embargo, estas pocas funciones constructoras enumeradas también se pueden usar para crear datos primitivos, no solo objetos (esta no es una característica general, sino solo para las funciones enumeradas). Las siguientes funciones devolverán datos primitivos de un tipo determinado:

`Boolean`, `Number`, `BigInt` y `String`.

La mayoría de estas funciones se pueden llamar sin argumentos. En tal situación:

- La función `String` por defecto creará y devolverá una cadena vacía primitiva: `""`.
- La función `Number` por defecto creará y devolverá el valor 0.
- La función `Boolean` por defecto creará y devolverá el valor de false.

```
1 const str = String();
2 const num = Number();
3 const bool = Boolean();
4
5 console.log(str); // ->
6 console.log(num); // -> 0
7 console.log(bool); // -> false
8
9 const big1 = BigInt(42);
10 console.log(big1); // -> 42n
11
12 const big2 = BigInt(); // -> Uncaught TypeError: Cannot convert undefined to a BigInt
13
14
15 |
```

app.js

Console >...

0  
false  
42  
Uncaught TypeError: Cannot convert undefined to a BigInt

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Conversiones de Tipo de Datos

### Funciones de construcción primitivas

Usar literales no es la única manera de crear variables de los tipos primitivos dados. La segunda opción es crearlos usando funciones del tipo **constructor**. Este tipo de funciones se utilizan principalmente en JavaScript para la programación orientada a objetos, que está fuera del alcance de nuestro curso. Sin embargo, estas pocas funciones constructoras enumeradas también se pueden usar para crear datos primitivos, no solo objetos (esta no es una característica general, sino solo para las funciones enumeradas). Las siguientes funciones devolverán datos primitivos de un tipo determinado:

`Boolean`, `Number`, `BigInt` y `String`.

La mayoría de estas funciones se pueden llamar sin argumentos. En tal situación:

- La función `String` por defecto creará y devolverá una cadena vacía primitiva: `""`.
- La función `Number` por defecto creará y devolverá el valor 0.
- La función `Boolean` por defecto creará y devolverá el valor de false.

```
1 const str = String();
2 const num = Number();
3 const bool = Boolean();
4
5 console.log(str); // ->
6 console.log(num); // -> 0
7 console.log(bool); // -> false
8
9 const big1 = BigInt(42);
10 console.log(big1); // -> 42n
11
12
13
14
15 |
```

app.js

Console >...

0  
false  
42

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Conversiones

Es una situación bastante común tener un valor de un tipo pero necesitar un valor de otro tipo. El ejemplo más simple es cuando tenemos un número, pero necesitamos agregarlo a algún texto. Las conversiones en JavaScript ocurren automáticamente en situaciones específicas, pero también se pueden usar explícitamente a través de funciones como `String()` o `Number()`. Anteriormente vimos cómo esas funciones podrían usarse para crear valores predeterminados de esos tipos, pero eso no es todo lo que pueden hacer. Esas funciones también aceptan argumentos entre paréntesis y (si es posible) los convertirán a un tipo dado.

```
const num = 42;

const strFromNum1 = String(num);
const strFromNum2 = String(8);
const strFromBool = String(true);
const numFromStr = Number("312");
const boolFromNumber = Boolean(0);
```

```
1 const num = 42;
2
3 const strFromNum1 = String(num);
4 const strFromNum2 = String(8);
5 const strFromBool = String(true);
6 const numFromStr = Number("312");
7 const boolFromNumber = Boolean(0);
8
9 console.log(strFromNum1); // -> "42"
10 console.log(strFromNum2); // -> "8"
11 console.log(strFromBool); // -> "true"
12 console.log(numFromStr); // -> 312
13 console.log(boolFromNumber); // -> false
14 |
```

app.js

Console >...

42  
8  
true  
312  
false

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Conversión a String

Las conversiones son las más fáciles de entender, ya que intentan cambiar directamente el valor a una cadena, y esto se puede hacer para todos los tipos de datos primitivos. Así que no hay sorpresas allí. Toma en cuenta que en el ejemplo, utilizamos la técnica descrita recientemente de la **interpolación de cadenas de caracteres**.

```
let str = "text";
let strStr = String(str);
console.log(`\$${typeof str} : ${str}`); // -> string : text
console.log(`\$${typeof strStr} : ${strStr}`); // -> string : t

let nr = 42;
let strNr = String(nr);
console.log(`\$${typeof nr} : ${nr}`); // -> number : 42
console.log(`\$${typeof strNr} : ${strNr}`); // -> string : 42

let bl = true;
let strBl = String(bl);
console.log(`\$${typeof bl} : ${bl}`); // -> boolean : true
console.log(`\$${typeof strBl} : ${strBl}`); // -> string : t
```

app.js

```
Console>_
string:text
string:text
number:42
string:42
boolean:true
string:true
bigint:123
string:123
```

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

## Conversión a Number

La conversión a un número no es tan obvia como la conversión a una cadena. Funciona como se esperaba para cadenas que representan números reales, como "14", "-72.134", o cadenas que representan números en notación científica, como "2e3", o los valores numéricos especiales como "NaN" o "Infinity".

Sin embargo, la cadena también puede contener números en formato hexadecimal, octal y binario. Deben estar precedidos por 0x, 0o o 0b respectivamente. Para cualquier cadena que no se pueda convertir a un valor especial, se devuelve `NaN` (no un número). Un `BigInt` también se puede convertir en un `Number`, pero debemos recordar que un `BigInt` puede almacenar valores mucho más grandes que un `Number`, por lo que para valores grandes, parte de ellos puede ser truncado o terminar siendo impreciso. El Boolean `true` se convierte en `1` y `false` en `0`; esto es común para muchos lenguajes de programación. Un intento de convertir un valor undefined dará como resultado `NaN`, mientras que null se convertirá en `0`.

```
console.log(Number(42)); // -> 42
```

app.js

```
Console>_
42
11
17
9
3
12000
Infinity
NaN
```

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

## Conversión a Boolean

Las conversiones a Boolean siguen reglas simples, ya que solo podemos tener uno de dos valores: `true` o `false`. El valor `false` siempre se devuelve para:

- `0`,
- `NaN`,
- cadena vacía,
- `undefined`,
- `null`

Cualquier otro valor dará como resultado que se devuelva `true`.

```
console.log(Boolean(true)); // -> true
console.log(Boolean(42)); // -> true
console.log(Boolean(0)); // -> false
console.log(Boolean(NaN)); // -> false

console.log(Boolean("texto")); // -> true
console.log(Boolean(""))); // -> false
```

app.js

```
Console>_
true
true
false
false
true
false
false
false
```

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

## Conversión a BigInt

Para que las conversiones a BigInt tengan éxito, necesitamos un número o una cadena que represente un número como un valor a convertir. Los valores para la conversión se pueden proporcionar en forma decimal predeterminada, así como en forma hexadecimal, octal o binaria. Esto se aplica tanto a la situación en la que damos los literales Number y String como argumentos (o variables que contienen datos de esos tipos). También podemos usar la notación exponencial, pero solo para argumentos Number. A diferencia de otras conversiones, la conversión a BigInt generará un error y detendrá el programa cuando no pueda convertir un valor dado.

**Nota:** Al probar el siguiente ejemplo, presta atención al hecho de que el primer error impide la ejecución de más código. Así que ejecuta el ejemplo varias veces seguidas, eliminando las llamadas incorrectas una por una.

```
console.log(BigInt(11)); // -> 1ln
console.log(BigInt(0x11)); // -> 17n
console.log(BigInt(11e2)); // -> 1100n
```

```
console.log(BigInt(true)); // -> ln
```

```
2 console.log(BigInt(0x11)); // -> 17n
3 console.log(BigInt(11e2)); // -> 1100n
4
5 console.log(BigInt(true)); // -> ln
6
7 console.log(BigInt("11")); // -> 1ln
8 console.log(BigInt("0x11")); // -> 17n
9
10 console.log(BigInt(null)); // -> Uncaught TypeError: Cannot convert null to a BigInt
11
12 console.log(BigInt(undefined)); // -> Uncaught TypeError: Cannot convert undefined to a BigInt
13
14 console.log(BigInt(NaN)); // -> Uncaught RangeError: The number NaN cannot be converted to a BigInt
15
16
17
```

app.js

Console >...

11

17

1100

1

11

17

Uncaught TypeError: Cannot convert null to a BigInt

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Conversión a BigInt

Para que las conversiones a BigInt tengan éxito, necesitamos un número o una cadena que represente un número como un valor a convertir. Los valores para la conversión se pueden proporcionar en forma decimal predeterminada, así como en forma hexadecimal, octal o binaria. Esto se aplica tanto a la situación en la que damos los literales Number y String como argumentos (o variables que contienen datos de esos tipos). También podemos usar la notación exponencial, pero solo para argumentos Number. A diferencia de otras conversiones, la conversión a BigInt generará un error y detendrá el programa cuando no pueda convertir un valor dado.

**Nota:** Al probar el siguiente ejemplo, presta atención al hecho de que el primer error impide la ejecución de más código. Así que ejecuta el ejemplo varias veces seguidas, eliminando las llamadas incorrectas una por una.

```
console.log(BigInt(11)); // -> 1ln
console.log(BigInt(0x11)); // -> 17n
console.log(BigInt(11e2)); // -> 1100n
```

```
console.log(BigInt(true)); // -> ln
```

```
1 console.log(BigInt(11)); // -> 1ln
2 console.log(BigInt(0x11)); // -> 17n
3 console.log(BigInt(11e2)); // -> 1100n
4
5 console.log(BigInt(true)); // -> ln
6
7 console.log(BigInt("11")); // -> 1ln
8 console.log(BigInt("0x11")); // -> 17n
9
10
11
12
13
```

app.js

Console >...

11

17

1100

1

11

17

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Conversiones Implícitas

Las conversiones también pueden ocurrir automáticamente y ocurren todo el tiempo. Este ejemplo simple lo demostrará (probamos un ejemplo similar cuando discutimos el tipo de dato String):

```
const str1 = 42 + "1";
console.log(str1);           // -> 421
console.log(typeof str1);    // -> string

const str2 = 42 - "1";
console.log(str2);           // -> 41
console.log(typeof str2);    // -> number
```

```
1 const str1 = 42 + "1";
2 console.log(str1);           // -> 421
3 console.log(typeof str1);    // -> string
4
5 const str2 = 42 - "1";
6 console.log(str2);           // -> 41
7 console.log(typeof str2);    // -> number
8
9
10
```

app.js

Console >...

421

string

41

number

Sandbox

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

[Sandbox](#)

## Tareas

### Datos Primitivos

- Escribe un fragmento de código que creará variables y las inicializará con valores Boolean, Number, BigInt, String y tipos undefined utilizando (siempre que sea posible) literales y funciones constructoras.
- Imprime todos los valores y todos los tipos de esos valores usando `console.log`. Intenta usar la interpolación de cadenas para mostrar el valor y el tipo al mismo tiempo con una sola llamada a `console.log`, por ejemplo, en el siguiente formato: 1000 [número].
- Realiza una cadena de conversiones: crea un Boolean a partir de un BigInt creado a partir de un Number que se creó a partir de un String. Comienza con el valor "1234". ¿Es posible?
- Intenta agregar dos valores del mismo tipo y verifica el tipo de resultado. Pruébalo para todos los tipos de datos primitivos.
- Intenta sumar dos valores de diferentes tipos y verifica los resultados.
- Intenta modificar la línea const str1 = 42 + "1"; para obtener el resultado 43 (sin eliminar las comillas alrededor del 1)

```

17 const numToInt = BigInt(strToInt);
18 const boolFromBigInt = Boolean(numToInt);
19 console.log(` ${boolFromBigInt} [${typeof boolFromBigInt}]`);
20
21 // Suma de dos valores del mismo tipo y verificación del tipo de resultado
22 const sumNumbers = numberVar + 10;
23 console.log(` ${sumNumbers} [${typeof sumNumbers}]`);
24
25 // Suma de dos valores de diferentes tipos y verificación del resultado
26 const sumDifferentTypes = numberVar + booleanVar;
27 console.log(` ${sumDifferentTypes} [${typeof sumDifferentTypes}]`);
28
29

```

app.js

Console >...

```

true [boolean]
42 [number]
1234 [bigint]
Hola [string]
undefined [undefined]
true [boolean]
52 [number]
43 [number]
43 [string]

```

[Fullscreen](#)

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

[Sandbox](#)

## Tareas

### Tarea 1

Escribe un código que cree variables y las inicialice con valores Boolean, Number, BigInt, String y tipos undefined usando (cuando sea posible) literales y funciones constructoras.

[Solución](#)

```

13 let ui = undefined;
14
15 console.log(` ${b1} [${typeof b1}]`);
16 console.log(` ${b2} [${typeof b2}]`);
17
18 console.log(` ${n1} [${typeof n1}]`);
19 console.log(` ${n2} [${typeof n2}]`);
20
21 console.log(` ${b1} [${typeof b1}]`);
22 console.log(` ${b2} [${typeof b2}]`);
23
24 console.log(` ${s1} [${typeof s1}]`);
25 console.log(` ${s2} [${typeof s2}]`);
26
27 console.log(` ${ui} [${typeof ui}]`);
28

```

app.js

Console >...

```

true [boolean]
true [boolean]
100 [number]
200 [number]
100 [bigint]
200 [bigint]
Hello [string]
Hello [string]

```

[Fullscreen](#)

### Tarea 2

Imprime todos los valores y todos los tipos de esos valores usando `console.log`. Intenta usar la interpolación de cadenas para mostrar el valor y el tipo al mismo tiempo con una sola llamada a `console.log`.

[Solución](#)

```

13 let s1 = "Hello";
14 let s2 = String("Hello");
15
16 let ui; // No es necesario inicializar con undefined, ya que es el valor por defecto.
17
18 console.log(` ${b1} [${typeof b1}]`);
19 console.log(` ${b2} [${typeof b2}]`);
20 console.log(` ${n1} [${typeof n1}]`);
21 console.log(` ${n2} [${typeof n2}]`);
22 console.log(` ${b1} [${typeof b1}]`);
23 console.log(` ${b2} [${typeof b2}]`);
24 console.log(` ${s1} [${typeof s1}]`);
25 console.log(` ${s2} [${typeof s2}]`);
26 console.log(` ${ui} [${typeof ui}]`);
27

```

app.js

Console >...

```

true [boolean]
true [boolean]
100 [number]
200 [number]
100 [bigint]
200 [bigint]
Hello [string]
Hello [string]

```

[Fullscreen](#)

### Tarea 3

Realizar una cadena de conversiones: crear un Boolean a partir de un BigInt creado a partir de un Number que se creó a partir de un String. Comienza con el valor "1234". ¿Es posible?

[Solución](#)

```

9
10 let s1 = "Hello";
11 let s2 = String("Hello");
12
13 let ui; // No es necesario inicializar con undefined, ya que es el valor por defecto.
14
15 console.log(` ${b1} [${typeof b1}]`);
16 console.log(` ${b2} [${typeof b2}]`);
17 console.log(` ${n1} [${typeof n1}]`);
18 console.log(` ${n2} [${typeof n2}]`);
19 console.log(` ${b1} [${typeof b1}]`);
20 console.log(` ${b2} [${typeof b2}]`);
21 console.log(` ${s1} [${typeof s1}]`);
22 console.log(` ${s2} [${typeof s2}]`);
23 console.log(` ${ui} [${typeof ui}]`);
24

```

app.js

Console >...

```

true [boolean]
true [boolean]
100 [number]
200 [number]
100 [bigint]
200 [bigint]
Hello [string]
Hello [string]

```

[Fullscreen](#)

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

Solución
Sandbox

el valor y el tipo al mismo tiempo con una sola llamada a `console.log`.

```
1 // Primer bloque de código
2 let booleanResult1 = Boolean(BigInt(Number("1234")));
3 console.log(`booleanResult1 [${typeof booleanResult1}]`); // Salida: true [boolean]
4
5 // Segundo bloque de código
6 let s = "1234";
7 let n = Number(s);
8 let bi = BigInt(n);
9 let booleanResult2 = Boolean(bi);
10 console.log(`booleanResult2 [${typeof booleanResult2}]`); // Salida: true [boolean]
11
```

**Tarea 3**

Realizar una cadena de conversiones: crear un `Boolean` a partir de un `BigInt` creado a partir de un `Number` que se creó a partir de un `String`. Comienza con el valor `"1234"`. ¿Es posible?

**Ejemplo**

```
app.js
Console>_
true [boolean]
true [boolean]
```

**Tarea 4**

Intenta agregar dos valores del mismo tipo y verifica el tipo de resultado. Pruébalo para todos los tipos primitivos.

**Ejemplo**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

Ejemplo
Sandbox

Intenta agregar dos valores del mismo tipo y verifica el tipo de resultado. Pruébalo para todos los tipos primitivos.

**Tarea 4**

Intenta agregar dos valores del mismo tipo y verifica el tipo de resultado. Pruébalo para todos los tipos primitivos.

**Ejemplo**

```
app.js
Console>_
1 [number]
300 [number]
300 [bigint]
Hello [string]
NaN [number]
```

**Tarea 5**

Prueba sumar dos valores de diferentes tipos y verifica los resultados.

**Ejemplos**

**Tarea 6**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

Ejemplo
Sandbox

Pruébalo para todos los tipos primitivos.

**Tarea 5**

Prueba sumar dos valores de diferentes tipos y verifica los resultados.

**Ejemplos**

**Tarea 6**

Intenta modificar la línea `const str1 = 42 + "1";` para obtener el resultado `43` (sin eliminar las comillas alrededor del `1`).

**Ejemplo**

```
app.js
Console>_
101 [number]
true100 [string]
101 [number]
100200 [string]
100200 [string]
100200 [string]
100200 [string]
100true [string]
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**JS INSTITUTE**  
Open Education & Development Group

« 2.2.1.22 Tipos de Datos - Tareas »

**Tarea 5**

Prueba sumar dos valores de diferentes tipos y verifica los resultados.

**Ejemplos**

**Tarea 6**

Intenta modificar la línea `const str1 = 42 + "1";` para obtener el resultado `43` (sin eliminar las comillas alrededor del `1`).

**Ejemplo**

```
const str1 = 42 + +"1";
```

app.js

```
1 const str1 = 42 + "1";
2 console.log(str1); // Resultado: 43
3
```

Console >...

```
43
```

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

This screenshot shows a JS Institute sandbox interface. It features a top navigation bar with the institute's logo and the title '2.2.1.22 Tipos de Datos - Tareas'. Below this are two sections: 'Tarea 5' and 'Tarea 6'. 'Tarea 5' contains a task to sum different data types and verify results. 'Tarea 6' is a challenge to modify a line of code to produce a specific result. On the left, there's a 'Ejemplos' (Examples) button and a code editor with a snippet for 'Tarea 6'. On the right, there's a code editor for 'app.js' with the modified line, a terminal window showing the output '43', and a 'Fullscreen' button. A cookie consent banner at the bottom states: 'We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#)'. A close button 'X' is also present.

## Modulo 2

### Sección 3

**JS INSTITUTE**  
Open Education & Development Group

« 2.3.1.2 Tipos de Datos Complejos »

**Tipos de datos complejos**

Limitaremos la discusión de tipos complejos a solo dos de ellos: objetos y arreglos. Desafortunadamente, incluso estos tipos deberán presentarse de manera simplificada. Esto debería ser suficiente para usarlos en su ámbito básico, pero las técnicas más avanzadas relacionadas con ellos, así como otros tipos complejos, se presentarán en las siguientes partes del curso.

**Objeto**

Los objetos tienen muchas aplicaciones en JavaScript. Una de las más básicas, y la única que usaremos ahora, es utilizarlo como una estructura conocida en informática como registro. Un **registro** es una colección de campos con nombre. Cada campo tiene su propio nombre (o clave) y un valor asignado. En el caso de los objetos de JavaScript, estos campos suelen denominarse propiedades. Los registros, o en nuestro caso, los objetos, te permiten almacenar múltiples valores de diferentes tipos en un solo lugar. En JavaScript, hay algunas formas de crear objetos, pero la más fácil y rápida es usar el literal de llaves {}.

```
let testObj = {};
console.log(typeof testObj); // -> object
```

app.js

```
1 // Crear un objeto con campos de nombre (o propiedades)
2 const persona = {
3   nombre: "Juan",
4   edad: 30,
5   ciudad: "Madrid",
6   casado: false
7 };
8
9 // Acceder a las propiedades del objeto
10 console.log(persona.nombre); // Resultado: "Juan"
11 console.log(persona.edad); // Resultado: 30
12 console.log(persona.ciudad); // Resultado: "Madrid"
13 console.log(persona.casado); // Resultado: false
14
```

Console >...

```
Juan
30
Madrid
false
```

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

This screenshot shows a JS Institute sandbox interface. It features a top navigation bar with the institute's logo and the title '2.3.1.2 Tipos de Datos Complejos'. Below this is a section titled 'Tipos de datos complejos' with a detailed explanation of objects. It includes a 'Objeto' section with a paragraph about objects being used as records. Below this is a code editor with a snippet for creating an object named 'testObj' and logging its type. To the right is a code editor for 'app.js' showing a script to create a 'persona' object with properties like 'nombre', 'edad', 'ciudad', and 'casado', and then log these properties. A terminal window on the right shows the expected output: 'Juan', '30', 'Madrid', and 'false'. A 'Fullscreen' button is also present. A cookie consent banner at the bottom states: 'We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#)'. A close button 'X' is also present.

 Sandbox

¿Para qué necesitamos objetos? La razón más simple para usarlos puede ser el deseo de almacenar varios valores en un solo lugar, que están vinculados entre sí por alguna razón.

Supongamos que recopilamos información sobre los usuarios de nuestro sistema. La información sobre un solo usuario consistirá en su nombre, apellido, edad y dirección de correo electrónico. Intentemos escribir un fragmento de código apropiado para dos usuarios, sin usar objetos por ahora.

```
let name1 = "Calvin";
let surname1 = "Hart";
let age1 = 66;
let email1 = "CalvinMHart@teleworm.us";

let name2 = "Mateus";
let surname2 = "Pinto";
let age2 = 21;
let email2 = "MateusPinto@dayrep.com";
```

```
7 // Imprimir el valor de la propiedad "phone" (que aún no existe)
8 console.log(user.phone); // -> undefined
10
11 // Agregar una nueva propiedad al objeto
12 user.phone = "904-399-7557";
13
14 // Imprimir el valor de la propiedad "phone"
15 console.log(user.phone); // -> 904-399-7557
16
17 // Eliminar la propiedad "phone" del objeto
18 delete user.phone;
19
20 // Imprimir el valor de la propiedad "phone" después de eliminarla (debe ser undefined)
21 console.log(user.phone); // -> undefined
22 |
```

app.js

Console>...
undefined
904-399-7557
undefined

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

## Arreglos

Un **arreglo**, como un objeto, es un tipo de datos complejo que se puede usar para almacenar una colección de datos. Similar a un objeto, los datos almacenados (los valores) pueden ser de cualquier tipo. La diferencia entre estas estructuras es que en un arreglo solo almacenamos valores, sin los nombres asociados (es decir, las claves).

Entonces, ¿cómo sabemos a qué elemento del arreglo nos referimos si no podemos señalarlo por su nombre? Lo sabemos porque los elementos del arreglo están ordenados (pero no necesariamente clasificados) y ocupan posiciones consecutivas y numeradas dentro de él. El número del campo donde se encuentra un valor particular en el arreglo se denomina índice o posición. El índice comienza desde 0.

La forma más sencilla de crear arreglos en JavaScript es usar corchetes (es un literal de arreglo). De esta forma, podemos crear tanto un arreglo vacío, en el que se insertarán los elementos más tarde, como un arreglo que contenga algunos elementos iniciales (que estarán separados por comas). Al referirnos a un elemento del arreglo en particular, usamos la notación de corchetes: después del nombre de la variable del arreglo, escribimos un corchete, en el que ponemos el índice del elemento que nos interesa.

```
1 let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
2 console.log(days[0]); // -> Sun
3 console.log(days[2]); // -> Tue
4 console.log(days[5]); // -> Fri
5
6 days[0] = "Sunday";
7 console.log(days[0]); // -> Sunday
8
9 let emptyArray = [];
10 console.log(emptyArray[0]); // -> undefined
11
12
13 |
```

app.js

Console>...
Sun
Tue
Fri
Sunday
undefined

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

¿Cómo podemos agregar un nuevo elemento a un arreglo existente, por ejemplo, uno vacío?

La forma más sencilla sería asignar un nuevo valor a una posición específica utilizando la notación de corchetes. Para el intérprete, no importa si ya hay algo en este índice o no. Simplemente coloca un nuevo valor allí. Lo interesante es que no tenemos que llenar el arreglo con elementos uno por uno; puedes dejar espacios vacíos en el arreglo.

```
let animals = [];
console.log(animals[0]); // -> undefined

animals[0] = "dog";
animals[2] = "cat";

console.log(animals[0]); // -> dog
console.log(animals[1]); // -> undefined
console.log(animals[2]); // -> cat
```

```
1 let names = [["Dalia", "Citzlali", "Andria", "Salvador"], ["William", "James", "Daniel"]];
2 console.log(names[0]); // -> ["Dalia", "Citzlali", "Andria", "Salvador"]
3 console.log(names[1]); // -> Citzlali
4 console.log(names[1][1]); // -> James
5
6 let femaleNames = names[0];
7 console.log(femaleNames[0]); // -> Dalia
8 console.log(femaleNames[2]); // -> Andria
9 |
```

app.js

Console>...
Dalia,Citzlali,Andria,Salvador
Citzlali
James
Dalia
Andria

 Fullscreen

En el ejemplo, declaramos un arreglo vacío llamado animals. Luego

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

### ¿Para qué pueden ser útiles los arreglos en la práctica?

Son principalmente una forma conveniente de almacenar una colección de elementos bajo un nombre. Además, es muy importante que podamos agregar nuevos elementos a un arreglo mientras el programa se está ejecutando.

¿Recuerdas el ejemplo con los usuarios del sistema que probamos mientras discutímos los objetos? Una de las desventajas de la solución presentada fue la necesidad de declarar variables para todos los usuarios, por lo que en la etapa de escritura del programa teníamos que saber el número de usuarios. Usando un arreglo, podemos agregar nuevos usuarios mientras se ejecuta el programa. Mencionamos varias veces que los elementos de un arreglo pueden ser cualquier dato, incluidos los objetos. Como recordatorio, repitamos el ejemplo en el que declaramos dos variables del tipo objeto, `user1` y `user2`, que contienen información sobre dos usuarios del sistema:

```
let user1 = {
  name: "Calvin",
  ...
}
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

```
15 } else {
16   console.log("El usuario en la posición 0 está indefinido.");
17 }
18
19 if (users[1]) {
20   console.log(users[1].name); // -> Mateus
21 } else {
22   console.log("El usuario en la posición 1 está indefinido.");
23 }
24
25 if (users[2]) {
26   console.log(users[2].name); // -> Irene
27 } else {
28   console.log("El usuario en la posición 2 está indefinido.");
29 }
30 }
```

app.js

Console >...

El usuario en la posición 0 está indefinido.  
El usuario en la posición 1 está indefinido.

 Sandbox

 Fullscreen

Ahora hagamos un pequeño experimento y apliquemos el operador `typeof` a la variable que contiene el arreglo. El resultado puede ser algo sorprendente:

```
let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
console.log(typeof days); // -> object
```

Para hablar en general, en JavaScript, todo excepto los datos primitivos son objetos. Los **arreglos** también se tratan como un tipo especial de objeto. El operador `typeof` no distingue entre tipos de objetos (o más precisamente, clases), por lo que nos informa que la variable `days` contiene un objeto. Si queremos asegurarnos de que la variable contiene un arreglo, podemos hacerlo usando el operador `instanceof`, entre otros. Esta estrechamente relacionado con la programación orientada a objetos, de la que no hablaremos en este curso, pero por el momento solo necesitamos saber cómo usarlo en esta situación específica.

```
let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
let day = "Sunday";
```

```
1 let days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
2 let day = "Sunday";
3
4 console.log(typeof days); // -> object
5 console.log(typeof day); // -> string
6
7 console.log(days instanceof Array); // -> true
8 console.log(day instanceof Array); // -> false
9
10
11
```

app.js

Console >...

object  
string  
true  
false

 Sandbox

 Fullscreen

Recientemente presentamos los conceptos de **método** y **propiedad**.

Aparecieron cuando hablábamos del tipo String. Estas eran funciones y valores relacionados con un objeto específico. Ahora resulta que un arreglo se implementa como un objeto en JavaScript, por lo que probablemente también tenga sus métodos y propiedades. Y de hecho los tiene. Hay muchos métodos muy útiles que nos ayudan a trabajar con arreglos, como combinar arreglos, cortar elementos, ordenar o filtrar.

Solo veremos algunos de ellos ahora, porque muchos otros requieren que podamos crear nuestras propias funciones. Volveremos sobre algunos de ellos en el apartado dedicado a las funciones.

#### length

La propiedad `length` se utiliza para obtener información sobre la longitud (la cantidad de elementos) del arreglo (incluidas las posiciones vacías entre los elementos existentes).

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

names[5] = "Amelia";
console.log(names.length); // -> 5
```

```
1 let names = ["Dalia", "Citolali", "Andria", "Salvador"];
2 console.log(names.length); // -> 4
3
4 names[5] = "Amelia";
5 console.log(names.length); // -> 6
6
7 console.log(names); // -> ["Dalia", "Citolali", "Andria", "Salvador", undefined, "Amelia"]
8 console.log(names[3]); // -> Salvador
9 console.log(names[4]); // -> undefined
10 console.log(names[5]); // -> Amelia
11
```

app.js

Console >...

4  
6  
Dalia,Citolali,Andria,Salvador,Amelia  
Salvador  
undefined  
Amelia

 Sandbox

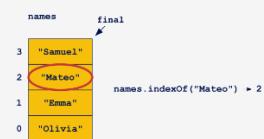
 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

### indexOf

El método `indexOf` se utiliza para buscar en el arreglo y localizar un valor dado. Si se encuentra el valor (el elemento está en el arreglo), se devolverá su índice (posición). El método devuelve -1 si no se encuentra el elemento. Si hay más de un elemento con el mismo valor en el arreglo, se devolverá el índice del primer elemento encontrado.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.indexOf("Mateo")); // -> 2
console.log(names.indexOf("Victor")); // -> -1
```



```
1 let names = ["Olivia", "Emma", "Mateo", "Samuel"];
2 console.log(names.indexOf("Mateo")); // -> 2
3 console.log(names.indexOf("Victor")); // -> -1
```

app.js

Console >...

2

-1

Sandbox

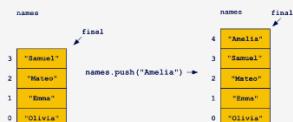
Fullscreen

### push

El método `push` coloca el elemento dado como su argumento al final del arreglo. La longitud del arreglo aumenta en 1 y el nuevo elemento se inserta a la derecha (tiene el índice más grande de todos los elementos).

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

names.push("Amelia");
console.log(names.length); // -> 5
console.log(names); // -> ["Olivia", "Emma", "Mateo", "Samuel", "Amelia"]
```



```
1 let names = ["Dalia", "Citlali", "Andria", "Salvador"];
2 console.log(names.length); // -> 4
3
4 names.push("Amelia");
5 console.log(names.length); // -> 5
6 console.log(names); // -> ["Dalia", "Citlali", "Andria", "Salvador", "Amelia"]
7
```

app.js

Console >...

4

5

Dalia,Citlali,Andria,Salvador,Amelia

Sandbox

Fullscreen

### unshift

El método `unshift` funciona de manera similar a `push`, con la diferencia de que se agrega un nuevo elemento al inicio del arreglo. La longitud del arreglo aumenta en 1, todos los elementos antiguos se mueven hacia la derecha y el nuevo elemento se coloca en el espacio vacío que se ha creado al inicio del arreglo. El índice del nuevo elemento es 0.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names);
console.log(names.indexOf("Mateo")); // -> 2
console.log(names.indexOf("Victor")); // -> -1 -> No existe
names.unshift("Amelia");
console.log(names.indexOf("Amelia")); // -> 0
console.log(names);
console.log(names);
```

```
1 let names = ["Dalia", "Citlali", "Andria", "Salvador"];
2 console.log(names);
3 console.log(names.indexOf("Andria")); // -> 2
4 console.log(names.indexOf("Victor")); // -> -1 -> No existe
5 names.unshift("Amelia");
6 console.log(names.indexOf("Amelia")); // -> 0
7 console.log(names);
8
```

app.js

Console >...

Dalia,Citlali,Andria,Salvador

2

-1

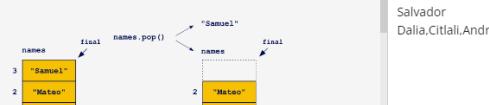
Sandbox

**pop**

El método `pop` te permite eliminar el último elemento del arreglo. Como resultado de su ejecución, se devuelve el elemento con mayor índice, mientras que al mismo tiempo se elimina del arreglo original. La longitud del arreglo obviamente se reduce en 1.

```
let names= ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

let name = names.pop();
console.log(names.length); // -> 3
console.log(name); // -> Samuel
console.log(names); // -> ["Olivia", "Emma", "Mateo"]
```



```
app.js
```

```
4
3
Salvador
Dalia,Citlali,Andria
```

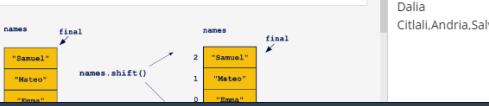
```
Sandbox
```

**shift**

El método `shift` funciona de manera similar a `pop`, solo que esta vez eliminamos el elemento del inicio del arreglo (con el índice 0). El método devuelve el elemento eliminado y todos los demás elementos se desplazan hacia la izquierda, llenando el espacio vacío. La longitud del arreglo original se reduce en 1.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
console.log(names.length); // -> 4

let name = names.shift();
console.log(names.length); // -> 3
console.log(name); // -> Olivia
console.log(names); // -> ["Emma", "Mateo", "Samuel"]
```



```
app.js
```

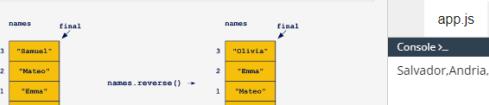
```
4
3
Dalia
Citlali,Andria,Salvador
```

```
Sandbox
```

**reverse**

El método `reverse` invierte el orden de los elementos de un arreglo. El método devuelve un arreglo con los elementos en orden inverso.

```
let names = ["Olivia", "Emma", "Mateo", "Samuel"];
names.reverse();
console.log(names); // -> ["Samuel", "Mateo", "Emma", "Olivia"]
```



```
app.js
```

```
Salvador,Andria,Citlali,Dalia
```

```
Sandbox
```

 Sandbox

### slice

El método `slice` te permite crear un nuevo arreglo a partir de elementos seleccionados del arreglo original. Llamar al método no afecta el arreglo original. El método toma uno o dos valores enteros como argumentos.

Las combinaciones básicas son:

- Un argumento mayor que cero: se copian todos los elementos del índice dado como argumento hasta el final del arreglo.
- Dos argumentos mayores que cero: se copia el elemento del índice especificado como primer argumento al elemento especificado como segundo argumento.
- Dos argumentos, el primero positivo, el segundo negativo: se copian todos los elementos desde el índice especificado hasta el final del arreglo, excepto el número especificado de los últimos elementos (por ejemplo, el argumento `-3` significa que no copiamos los últimos tres elementos).
- Un argumento negativo: el número especificado de los últimos elementos se copian al final del arreglo (por ejemplo, `-2` significa que se copian los dos últimos elementos).

```

1 let names = ["Dalia", "Citlali", "Adrian", "Salvador"];
2 let n1 = names.slice(2);
3 console.log(n1); // -> ["Adrian", "Salvador"]
4
5 let n2 = names.slice(1,3);
6 console.log(n2); // -> ["Citlali", "Adrian"]
7
8 let n3 = names.slice(0, -1);
9 console.log(n3); // -> ["Dalia", "Citlali", "Adrian"]
10
11 let n4 = names.slice(-1);
12 console.log(n4); // -> ["Salvador"]
13
14 console.log(names); // -> ["Dalia", "Citlali", "Adrian", "Salvador"]
15
16

```

app.js

Console&gt;\_

Adrian,Salvador  
Citlali,Adrian  
Dalia,Citlali,Adrian  
Salvador  
Dalia,Citlali,Adrian,Salvador

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 
 Sandbox

### concat

El método `concat` crea un nuevo arreglo adjuntando elementos del arreglo dado como argumento a los elementos originales del arreglo. El método no cambia ni el arreglo original ni el arreglo especificado como argumento.

```

let names = ["Olivia", "Emma", "Mateo", "Samuel"];
let otherNames = ["William", "Jane"];
let allNames = names.concat(otherNames);

console.log(names); // -> ["Olivia", "Emma", "Mateo", "Samuel"]
console.log(otherNames); // -> ["William", "Jane"]
console.log(allNames); // -> ["Olivia", "Emma", "Mateo", "Samuel", "William", "Jane"]

```

app.js

Console&gt;\_

Dalia,Citlali,Salvador  
William,Jane  
Dalia,Citlali,Salvador,William,Jane

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 
 Sandbox

### Tareas

#### Objetos

Crea un objeto que describa un boleto de tren y guárdalo en la variable `ticket`. El objeto debe tener tres campos:

- estación inicial (el nombre clave es `'from'`, y como valor, proporciona el nombre de la estación más cercana en tu área)
- estación final (el nombre clave es `'to'`, y como valor, dar cualquier otra estación dentro de 100 km)
- el precio del boleto (el nombre clave es `'price'`, y como valor, proporciona la cantidad que te gustaría pagar por este boleto)

El objeto debe crearse usando llaves {}, en los que todos los campos se enumerarán inmediatamente. Luego muestra los valores de todos los campos del ticket en la consola.

Declara un objeto vacío y guárdalo en la variable `person`. Usando la notación de punto, agrega los campos `firstName` y `lastName` al objeto ingresando tus datos como valores. Intenta mostrar los campos individuales en la consola.

```

54 // Copiar los dos últimos libros al nuevo arreglo usando slice
55 let lastTwoBooks = books.slice(-2);
56
57 // Eliminar el primer libro del arreglo
58 books.shift();
59
60 // Mostrar la longitud del arreglo de libros después de eliminar el primer libro
61 console.log("Número total de libros después de eliminar uno:", books.length);
62
63 // Mostrar los nombres de todos los libros en la colección después de eliminar el primer libro
64 books.forEach(book => console.log(book.title));
65
66

```

app.js

Console&gt;\_

Estación inicial: Estación A  
Estación final: Estación B  
Precio del boleto: 50  
Nombre: John  
Apellido: Doe  
Número total de libros: 4  
Speaking JavaScript  
Programming JavaScript Applications  
Understanding ECMAScript 6  
Learning JavaScript Design Patterns

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Tareas

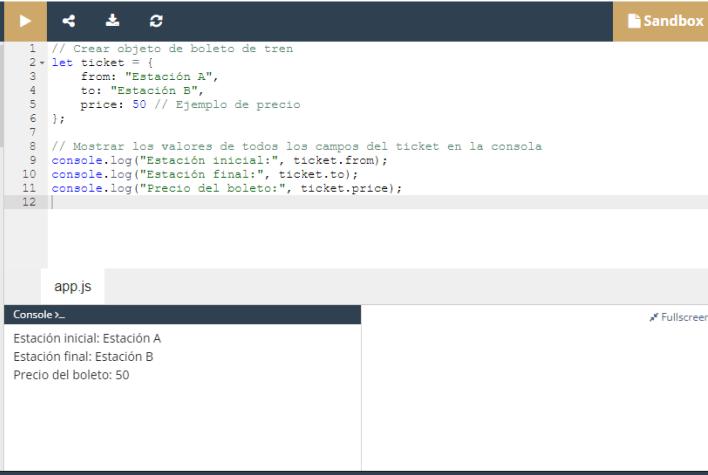
### Objetos

#### Tarea 1

Crea un objeto que describa un boleto de tren y guárdalo en la variable ticket. El objeto debe tener tres campos:

- estación inicial (el nombre clave es `from`, y como valor, proporciona el nombre de la estación más cercana en tu área)
- estación final (el nombre clave es `to`, y como valor, dar cualquier otra estación dentro de 100 km)
- el precio del boleto (el nombre clave es `price`, y como valor, proporciona la cantidad que te gustaría pagar por este boleto)  
El objeto debe crearse usando llaves {}, en los que todos los campos se enumerarán inmediatamente. Luego muestra los valores de todos los campos del ticket en la consola.

[Ejemplo](#)



```
1 // Crear objeto de boleto de tren
2 let ticket = {
3   from: "Estación A",
4   to: "Estación B",
5   price: 50 // Ejemplo de precio
6 };
7
8 // Mostrar los valores de todos los campos del ticket en la consola
9 console.log("Estación inicial:", ticket.from);
10 console.log("Estación final:", ticket.to);
11 console.log("Precio del boleto:", ticket.price);
12 
```

app.js

Console >...

Estación inicial: Estación A  
Estación final: Estación B  
Precio del boleto: 50

Fullscreen

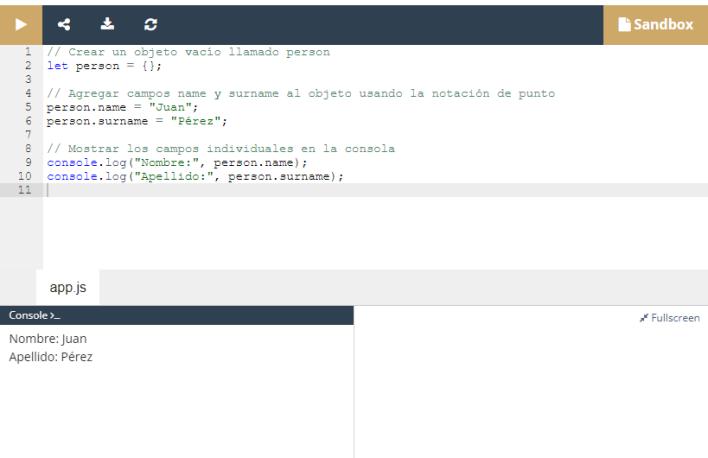
We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

[Ejemplo](#)

#### Task 2

Declara un objeto vacío y guárdalo en la variable person. Usando la notación de punto, agrega los campos `name` y `surname` al objeto. Ingresando tus datos como valores. Intenta mostrar los campos individuales en la consola.

[Ejemplo](#)



```
1 // Crear un objeto vacío llamado person
2 let person = {};
3
4 // Agregar campos name y surname al objeto usando la notación de punto
5 person.name = "Juan";
6 person.surname = "Pérez";
7
8 // Mostrar los campos individuales en la consola
9 console.log("Nombre:", person.name);
10 console.log("Apellido:", person.surname);
11 
```

app.js

Console >...

Nombre: Juan  
Apellido: Pérez

Fullscreen

### Arreglos

#### Tarea 3

Estamos creando una pequeña biblioteca de libros sobre programación en JavaScript. Tenemos tres libros y queremos preparar una lista de ellos. Almacenaremos tres datos de cada libro: el título, el autor y el número de páginas:

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

#### Tarea 3

Estamos creando una pequeña biblioteca de libros sobre programación en JavaScript. Tenemos tres libros y queremos preparar una lista de ellos. Almacenaremos tres datos de cada libro: el título, el autor y el número de páginas:

- *Speaking JavaScript*, Axel Rauschmayer, 460;
- *Programming JavaScript Applications*, Eric Elliott, 254;
- *Understanding ECMAScript 6*, Nicholas C. Zakas, 352.

- Crea un arreglo de tres objetos que representen los libros. Cada objeto debe tener las siguientes propiedades: title, author, pages

[Ejemplo](#)



```
6   },
7   {
8     title: "Programming JavaScript Applications",
9     author: "Eric Elliott",
10    pages: 254
11  },
12  {
13    title: "Understanding ECMAScript 6",
14    author: "Nicholas C. Zakas",
15    pages: 352
16  }
17 ];
18
19 // Mostrar los títulos de los libros en la consola
20 books.forEach(book => console.log(book.title));
21 
```

app.js

Console >...

Speaking JavaScript  
Programming JavaScript Applications  
Understanding ECMAScript 6

Fullscreen

#### Tarea 4

Agregar un nuevo libro a la colección: *Learning JavaScript Design Patterns*, por Addy Osmani, 254 páginas. Usa el método apropiado

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

 Ejemplo

**Tarea 4**

Agregar un nuevo libro a la colección: *Learning JavaScript Design Patterns*, por Addy Osmani, 254 páginas. Usa el método apropiado para adjunta el libro al final del arreglo. Muestra la longitud del arreglo y, a su vez, todos los nombres de los libros en la colección.

 Ejemplo

**Task 5**

Utiliza el comando slice para copiar los dos últimos libros al nuevo arreglo.

 Ejemplo

```

16     }
17   ];
18
19 // Agregar un nuevo libro al final del arreglo
20 books.push([
21   title: "Learning JavaScript Design Patterns",
22   author: "Addy Osmani",
23   pages: 254
24 ]);
25
26 // Mostrar la longitud del arreglo
27 console.log("Número total de libros:", books.length);
28
29 // Mostrar los títulos de todos los libros en la colección
30 books.forEach(book => console.log(book.title));
31

```

app.js

Console &gt;-

Número total de libros: 4  
SpeakingJavaScript  
Programming JavaScript Applications  
Understanding ECMAScript 6  
Learning JavaScript Design Patterns

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 
 Sandbox

**Task 5**

Utiliza el comando slice para copiar los dos últimos libros al nuevo arreglo.

 Ejemplo

**Tarea 6**

El primer libro de la colección se pierde en circunstancias inexplicables. Ya has aceptado la pérdida, así que ahora eliminalo del arreglo. ¿Cuál método usarás para este propósito? Muestra la longitud del arreglo y todos los nombres de los libros de la colección a su vez.

 Ejemplo

**Tarea 7**

```

13   },
14   title: "Understanding ECMAScript 6",
15   author: "Nicholas C. Zakas",
16   pages: 352
17 },
18 {
19   title: "Learning JavaScript Design Patterns",
20   author: "Addy Osmani",
21   pages: 254
22 };
23
24 // Copiar los dos últimos libros al nuevo arreglo
25 let lastTwoBooks = books.slice(-2);
26
27 console.log(lastTwoBooks);
28

```

app.js

Console &gt;-

[object Object],[object Object]

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 
 Sandbox

**arreglo.**
 Ejemplo

**Tarea 6**

El primer libro de la colección se pierde en circunstancias inexplicables. Ya has aceptado la pérdida, así que ahora eliminalo del arreglo. ¿Cuál método usarás para este propósito? Muestra la longitud del arreglo y todos los nombres de los libros de la colección a su vez.

 Ejemplo

**Tarea 7**

Muestra la suma de las páginas de todos los libros de la colección.

 Ejemplo

```

1 // Definir la colección de libros
2 let colecciónLibros = ["Libro 1", "Libro 2", "Libro 3", "Libro 4", "Libro 5"];
3
4 // Eliminar el primer libro de la colección
5 colecciónLibros.shift();
6
7 // Mostrar la longitud del arreglo
8 console.log("Longitud del arreglo después de eliminar el primer libro:", colecciónLibros.length);
9
10 // Mostrar los nombres de los libros uno por uno
11 console.log("Nombres de los libros de la colección:");
12 colecciónLibros.forEach(libro => {
13   console.log(libro);
14 });
15

```

app.js

Console &gt;-

Longitud del arreglo después de eliminar el primer libro: 4

Nombres de los libros de la colección:

Libro 2  
Libro 3  
Libro 4  
Libro 5

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

**Tarea 6**

El primer libro de la colección se pierde en circunstancias inexplicables. Ya has aceptado la pérdida, así que ahora elimínalo del arreglo. ¿Cuál método usarás para este propósito? Muestra la longitud del arreglo y todos los nombres de los libros de la colección a su vez.

**Ejemplo**

```
7 ];
8 // Función para calcular la suma de las páginas de los libros
9 function sumaPaginas(libros) {
10   let suma = 0;
11   for (let i = 0; i < libros.length; i++) {
12     suma += libros[i].paginas;
13   }
14   return suma;
15 }
16
17 // Calcular la suma de las páginas
18 const sumaTotal = sumaPaginas(libros);
19
20 console.log("La suma total de las páginas de todos los libros es: " + sumaTotal);
21
22
```

app.js

**Tarea 7**

Muestra la suma de las páginas de todos los libros de la colección.

**Ejemplo**

Console >\_ La suma total de las páginas de todos los libros es: 1485

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**LABORATORIO**

**Tiempo estimado**  
15-30 minutos

**Nivel de dificultad**  
Fácil

**Objetivos**  
Familiarizar al estudiante con:

- Las propiedades básicas de los tipos de datos complejos Array y Object (usados como un registro) y ser capaz de usarlos en la práctica.

**Escenario**

¿Recuerdas la lista de contactos que se creó mientras realizabas la tarea del laboratorio anterior? Tienes que admitir que a primera vista parecía bastante extraño. Tuvimos que usar nueve variables para almacenar información sobre solo tres usuarios. Lo que es peor, agregar cada nuevo usuario

```
6   name: "Raja Villarreal",
7   phone: "0866 398 2895",
8   email: "posuere.vulputate@sed.com"
9 ];
10  {
11    name: "Helen Richards",
12    phone: "0800 1111",
13    email: "libero@convallis.edu"
14  }];
15 contacts.forEach(contact => {
16   console.log("Name:", contact.name);
17   console.log("Phone:", contact.phone);
18   console.log("Email:", contact.email);
19   console.log("-----");
20 });
21
```

app.js

Console >\_ Name: Maxwell Wright  
Phone: (0191) 719 6495  
Email: Curabitur.egestas.nunc@nonummyac.co.uk  
-----  
Name: Raja Villarreal  
Phone: 0866 398 2895  
Email: posuere.vulputate@sed.com

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

## Modulo 2

### Sección 4

## Comentarios

Los comentarios son algo común en la programación. "Comentar" puede no ser una técnica de programación clave (si se le puede llamar técnica), pero te permite mejorar tu trabajo con el código, entre otras cosas, haciéndolo más legible. Entonces, ¿qué son los comentarios y por qué los necesitamos?

Los comentarios son solo texto sin formato, totalmente ignorados por el intérprete de JavaScript, que generalmente sirven para explicar una determinada pieza de código, que por algunas razones puede no ser completamente legible. Sin embargo, no podemos escribirlos con total libertad, ya que el intérprete intentará tratarlos como comandos, nombres de variables o palabras clave. Entonces JavaScript necesita distinguir los comentarios del resto del código. En JavaScript, tenemos dos tipos de comentarios, y ambos se usan comúnmente en muchos lenguajes de programación, desde la familia de lenguajes C hasta Python. Se denominan comentarios de una sola línea y de varias líneas.

### Comentarios de una sola línea

Esta es la forma principal de comentar el código. Utiliza un carácter de doble diagonal al comienzo del comentario que se extiende hasta el final de la línea.

```
1 // Definimos un arreglo de objetos que representan contactos
2 let listaContactos = [
3   { nombre: "Ana Martinez",
4     telefono: "(0191) 719 6495",
5     correo: "ana.martinez@example.com"
6   },
7   { nombre: "Pedro Garcia",
8     telefono: "0866 398 2895",
9     correo: "pedro.garcia@example.com"
}
```

app.js

```
Console>_
Nombre: Ana Martinez
Teléfono: (0191) 719 6495
Correo electrónico: ana.martinez@example.com

Nombre: Pedro García
Teléfono: 0866 398 2895
Correo electrónico: pedro.garcia@example.com

Nombre: Laura Fernández
Teléfono: 0800 1111
Correo electrónico: laura.fernandez@example.com
```

Sandbox

Fullscreen

## Comentarios de varias líneas

Los comentarios de varias líneas, también conocidos como comentarios de bloque, permiten que los comentarios abarquen varias líneas, pero también te permiten colocar comentarios dentro de una línea, lo que no es posible con los comentarios de una sola línea. Se crean con una diagonal y un asterisco al principio del comentario y un asterisco y una diagonal al final.

```
/*
  Este es un bloque
  de comentarios y puede
  abarcar varias líneas
  *
  Entonces este código no se ejecutará
  console.log("Hola, Mundo!");
*/
let x /* porque no hay mejor nombre */ = 42;
console.log(x);
```

app.js

```
Console>_
42
```

Sandbox

Fullscreen

## Resumen

Por el momento, el uso de comentarios puede parecerse una simple curiosidad. Recuerda que con el tiempo, comenzarás a escribir programas que se volverán cada vez más complejos. Los comentarios te brindan la oportunidad de aumentar la claridad del código agregando información que ayudará a otros a comprender partes seleccionadas del mismo. El siguiente paso será la capacidad de generar documentación de proyectos a partir de ellos, aunque para ello deberás ajustar tus comentarios a la convención que impone la herramienta seleccionada. Los comentarios también son muy útiles para activar y desactivar piezas seleccionadas de código, que usamos con mayor frecuencia cuando probamos versiones alternativas o cuando buscamos errores en ellas.

## Tarea

Hay un código que actualmente no funciona. Intenta arreglarlo usando solo comentarios. Intenta, si es posible, usar los atajos de teclado en tu editor para este propósito.

Revisar

```
2 const prefix = "username_";
3
4 let userName = "Jack";
5 // const userName = "Adam";
6
7 let prefixedUserName;
8 // const prefixedUserName;
9
10 userName = "John";
11 prefixedUserName = prefix + userName;
12
13 console.log(prefixedUserName /*+ prefixedUserName2*/);
14 // console.log(prefixedUserName2);
15
16
17 |
```

app.js index.html style.css

```
Console>_
username_John
```

Sandbox

Fullscreen



privacy and [cookie policies](#).'" data-bbox="136 107 856 345"/>

```

9 let prefixedUserName;
10 // Eliminamos la segunda declaración de prefixedUserName
11 // const prefixedUserName;
12
13 userName = "John";
14 prefixedUserName = prefix + userName;
15
16 console.log(prefixedUserName); // Imprimimos el userName prefijado
17
18 // Si querías imprimir una variable prefixedUserName2, asegúrate de definirla primero
19 // console.log(prefixedUserName2);
20
21 // Si deseas imprimir prefixedUserName2, necesitas definirlo primero
22 // let prefixedUserName2 = "something";
23 // console.log(prefixedUserName2);
24
    app.js
  
```

Console> username\_John

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Modulo 3

### Sección 1

privacy and [cookie policies](#)'." data-bbox="136 434 856 670"/>

```

1 let year = 2050;
2 year = 2051;
3 let newYear = year;
4
5 console.log(newYear); // El resultado será 2051
6
    app.js
  
```

Console> 2051

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## « 3.1.1.4 Operadores aritméticos »

 Sandbox

## Operadores aritméticos

Los operadores aritméticos expresan operaciones matemáticas y aceptan valores numéricos y variables. Todos los operadores aritméticos, excepto la suma, intentarán convertir implícitamente los valores al tipo Number antes de realizar la operación.

El operador de suma convertirá todo a String si alguno de los operandos es de tipo String, de lo contrario los convertirá a Number como el resto de los operadores aritméticos. El orden de las operaciones se respeta en JavaScript como en matemáticas, y podemos usar paréntesis como en matemáticas para cambiar el orden de las operaciones si es necesario.

En general, es un buen hábito usar paréntesis para forzar la precedencia y el orden de las operaciones, no solo las aritméticas. La precedencia de las operaciones realizadas por el intérprete no siempre será tan intuitiva como la precedencia de las operaciones aritméticas conocidas de las matemáticas.

```
console.log(2 + 2 * 2); // -> 6
console.log(2 + (2 * 2)); // -> 6
console.log((2 + 2) * 2); // -> 8
```

 app.js

 app.js

 app.js

 app.js

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## « 3.1.1.5 Operadores aritméticos unarios »

 Sandbox

## Operadores aritméticos unarios

También existen varios operadores aritméticos unarios (que operan en un solo operando). Entre ellos se encuentran los operadores de más (+) y menos (-).

Ambos operadores convierten los operandos al tipo Number, mientras que el operador de menos (negativo) lo niega.

```
let str = "123";
let n1 = +str;
let n2 = -str;
let n3 = -n2;
let n4 = +"abcd";

console.log(`$({str}) : ${typeof str}`); // -> 123 : string
console.log(`$({n1}) : ${typeof n1}`); // -> 123 : number
console.log(`$({n2}) : ${typeof n2}`); // -> -123 : number
console.log(`$({n3}) : ${typeof n3}`); // -> 123 : number
console.log(`$({n4}) : ${typeof n4}`); // -> NaN : number
```

 app.js

 app.js

 app.js

 app.js

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## « 3.1.1.6 Operadores unarios de incremento y decrecimiento »

 Sandbox

## Operadores unarios de incremento y decrecimiento

Entre los operadores aritméticos, también tenemos a nuestra disposición los operadores de **incremento** `++` y **decremento** `--` unario, tanto en versiones de prefijo como de sufijo. Nos permiten aumentar (incrementar) o disminuir (decrementar) el valor del operando en 1.

Estos operadores en la versión de sufijo (es decir, el operador está en el lado derecho del operando) realizan la operación cambiando el valor de la variable, pero devuelven el valor antes del cambio. La versión de prefijo del operador (es decir, el operador se coloca antes del operando) realiza la operación y devuelve el nuevo valor.

Probablemente, la forma más fácil de entenderlo es usar un ejemplo del editor.

Esto sucede porque la línea de código:

```
console.log(n1++);
```

se interpreta como:

 app.js

 app.js

 app.js

 app.js

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Operadores de Asignación Compuesta

Los operadores aritméticos binarios se pueden combinar con el operador de asignación, dando como resultado la asignación de suma `+=`, la asignación de resta `-=`, la asignación de la multiplicación `*=`, la asignación de la división `/=`, la asignación del residuo `%=` y la asignación de potencia `**=`.

Cada uno de estos operadores toma un valor de la variable a la que se va a realizar la asignación (el operando izquierdo) y lo modifica realizando una operación aritmética utilizando el valor del operando derecho. El nuevo valor se asigna al operando izquierdo. Por ejemplo, el siguiente fragmento de código:

```
x += 100;
```

puede ser escrito de la forma:

```
x = x + 100;
```

Por lo tanto, no debería ser difícil entender cómo funciona el siguiente ejemplo:

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

```
1 let x = 10;
2
3 x += 2;
4 console.log(x); // -> 12
5 x -= 4;
6 console.log(x); // -> 8
7 x *= 3;
8 console.log(x); // -> 24
9 x /= 6;
10 console.log(x); // -> 4
11 x **= 3;
12 console.log(x); // -> 64
13 x %= 10;
14 console.log(x); // -> 4
15
```

app.js

Console>...

```
12
8
24
4
64
4
```

 Fullscreen

## Operadores lógicos y valores no booleanos

Mientras los operandos sean del tipo Boolean, podemos ver fácilmente lo que se devolverá. Pero esos operadores también se pueden usar con diferentes tipos de datos. El caso más fácil es el NO lógico. Primero, el operando se convierte temporalmente a un valor booleano (según las reglas explicadas en el capítulo anterior) y solo entonces se trata con la acción de operador adecuada (es decir, un valor verdadero se convierte en falso y viceversa). Por lo tanto, el operador NOT siempre devolverá falso o verdadero. A menudo, se utiliza la doble negación para convertir cualquier tipo a Boolean.

```
let nr = 0;
let year = 1970;
let name = "Alice";
let empty = "";

console.log(!nr); // -> true
console.log(!year); // -> false
console.log(!name); // -> false
console.log(!empty); // -> true
```

```
1 console.log(true && 1991); // -> 1991
2 console.log(false && 1991); // -> false
3 console.log(2 && 5); // -> 5
4 console.log(0 && 5); // -> 0
5 console.log("Alice" && "Bob"); // -> Bob
6 console.log("") && "Bob"); // -> empty string
7
8
9 console.log(true || 1991); // -> true
10 console.log(false || 1991); // -> 1991
11 console.log(2 || 5); // -> 2
12 console.log(0 || 5); // -> 5
13 console.log("Alice" || "Bob"); // -> Alice
14 console.log("") || "Bob"); // -> Bob
15
```

app.js

Console>...

```
1991
false
5
0
Bob
true
1991
2
```

 Fullscreen

## Operadores lógicos y valores no booleanos - continuación

Ambos operadores también utilizan la evaluación de cortocircuito.

Entonces, si el primer operando de AND (Y) es `false`, se devolverá y no se realizará ninguna otra verificación.

Por el contrario, si el primer operando de OR (O) es `true`, se devolverá y no se realizará ninguna otra comprobación. Esto acelera la ejecución del código, pero tiene un efecto secundario visible en este ejemplo:

```
let x = 0;
let y = 0;
console.log(x++ && y++); // -> 0
console.log(x); // -> 1
console.log(y); // -> y == 0
```

```
1 let x = 3;
2 let y = 1;
3 console.log(x++ && y++); // -> 1
4 console.log(x); // -> 4
5 console.log(y); // -> 2
6
```

app.js

Console>...

```
1
4
2
```

 Fullscreen

La instrucción `y++` nunca se ejecutará, lo que puede resultar confuso.

Los operadores lógicos se suelen utilizar junto con los **operadores condicionales**, y son especialmente útiles en **instrucciones condicionales**.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Operadores de Asignación Compuesta

Al igual que los operadores aritméticos, los **operadores lógicos binarios** se pueden usar en combinación con el operador de asignación, creando una asignación AND lógica `&&=` y una asignación OR lógica `||=`.

Debería ser fácil imaginar cómo funcionan. En el caso del operador AND, podemos comprobarlo con el siguiente ejemplo:

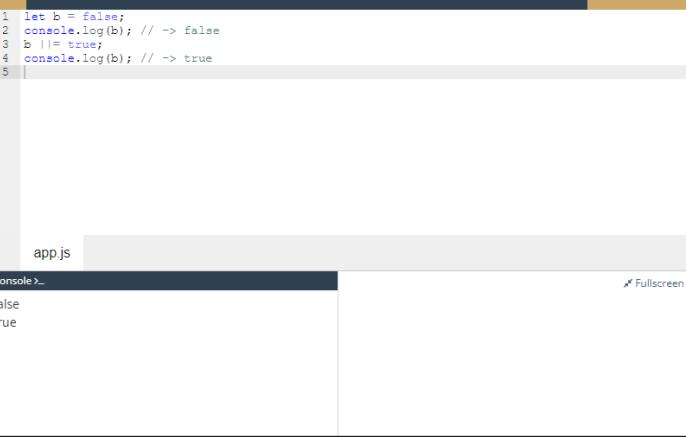
```
let a = true;
console.log(a); // -> true
a &&= false;
console.log(a); // -> false
```

La instrucción `a &&= false` significa exactamente lo mismo que `a = a && false`.

Podemos preparar un ejemplo similar para operaciones OR:

```
let b = false;
console.log(b); // -> false
b ||= true;
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X



```
1 let b = false;
2 console.log(b); // -> false
3 b ||= true;
4 console.log(b); // -> true
```

app.js      Console >...  
false  
true

## Tareas

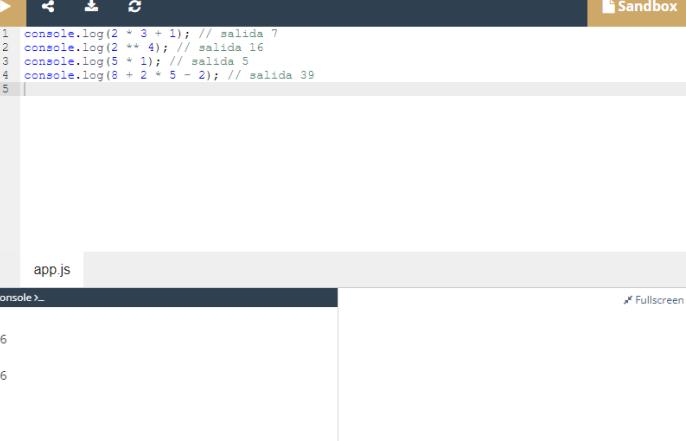
### Tarea 1

#### Operadores aritméticos

Completa los operadores que faltan para obtener el resultado esperado (reemplaza el guión bajo con el operador apropiado):

```
console.log(2 _ 3 _ 1); // salida 7
console.log(2 _ 4); // salida 16
console.log(5 _ 1); // salida 5
console.log(8 _ 2 _ 5 _ 2); // salida 39
```

**Ejemplo**



```
1 console.log(2 * 3 + 1); // salida 7
2 console.log(2 ** 4); // salida 16
3 console.log(5 * 1); // salida 5
4 console.log(8 + 2 * 5 - 2); // salida 39
```

app.js      Console >...  
7  
16  
5  
16

### Tarea 2

#### Operadores de comparación

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

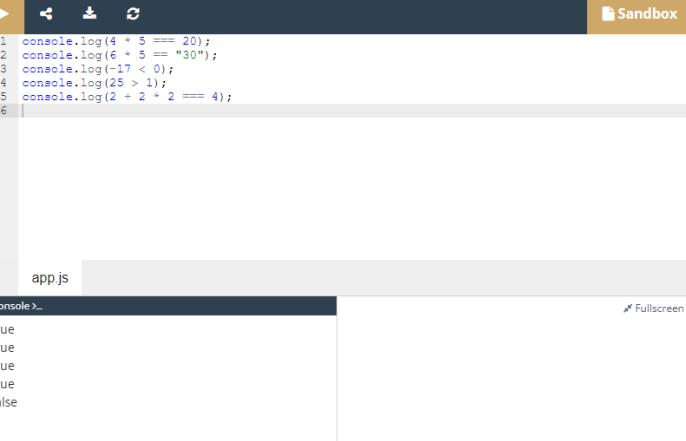
### Tarea 2

#### Operadores de comparación

Completa los operadores de comparación que faltan de tal manera que todas las expresiones resulten `true` - verdaderas (reemplaza el guión bajo con el operador apropiado):

```
console.log(4 + 5 _ 20);
console.log(6 * 5 _ "30");
console.log(-17 _ 0);
console.log(25 _ 1);
console.log(2 + 2 * 2 _ 4);
```

**Ejemplo**



```
1 console.log(4 + 5 === 20);
2 console.log(6 * 5 === "30");
3 console.log(-17 < 0);
4 console.log(25 > 1);
5 console.log(2 + 2 * 2 === 4);
```

app.js      Console >...  
true  
true  
true  
true  
false

### Tarea 3

#### Operadores Lógicos

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

« 3.1.1.12 Operadores - Tareas »

Ejemplo

**Tarea 3**

**Operadores Lógicos**

Completa los operadores de comparación que faltan de tal manera que todas las expresiones resulten `true` - verdaderas (reemplaza el guión bajo con el operador apropiado):

```
console.log(true _ false);
console.log(false _ false);
console.log(false _ false _ true);
console.log(true _ false _ false && true);
```

Ejemplo

app.js

Console>...

true  
false  
true  
false

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

## Modulo 3

### Sección 2

**JS INSTITUTE**  
Open Education & Development Group

« 3.2.1.2 Operadores de cadenas »

**Operadores de cadenas**

El único operador en este grupo es la **concatenación** `+`. Este operador convertirá todo a una **cadena** si alguno de los operandos es de tipo String. Finalmente, creará una nueva cadena de caracteres, adjuntando el operando derecho al final del operando izquierdo.

```
let greetings = "Hi";
console.log(greetings + " " + "Alice"); // -> Hi Alice

let sentence = "Happy New Year ";
let newSentence = sentence + 2003;

console.log(newSentence); // -> Happy New Year 2003
console.log(typeof newSentence); // -> string
```

**Operadores de asignación compuesta**

Probablemente puedas adivinar que este operador también se puede usar junto con el operador de reemplazo. Su funcionamiento es tan intuitivo que solo verás un ejemplo sencillo:

app.js

Console>...

Happy New Year 2003

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

[Sandbox](#)

## Operadores de cadenas

El único operador en este grupo es la **concatenación** `+`. Este operador convertirá todo a una **cadena** si alguno de los operandos es de tipo String. Finalmente, creará una nueva cadena de caracteres, adjuntando el operando derecho al final del operando izquierdo.

```
let greetings = "Hi";
console.log(greetings + " " + "Alice"); // -> Hi Alice

let sentence = "Happy New Year ";
let newSentence = sentence + 2003;

console.log(newSentence); // -> Happy New Year 2003
console.log(typeof newSentence); // -> string
```

```
1 let greetings = "Hi";
2 console.log(greetings + " " + "Brayan"); // -> Hi Alice
3
4 let sentence = "Happy New Year ";
5 let newSentence = sentence + 2003;
6
7 console.log(newSentence); // -> Happy New Year 2003
8 console.log(typeof newSentence); // -> string
9
10
11
```

app.js

Console>\_
  
Hi Brayan
  
Happy New Year 2003
  
string

[Fullscreen](#)

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). [X](#)

## Operadores de asignación compuesta

Probablemente puedas adivinar que este operador también se puede usar junto con el operador de reemplazo. Su funcionamiento es tan intuitivo que solo necesitas un ejemplo:

[Sandbox](#)

## Operadores de comparación - continuación

También tenemos operadores que nos permiten comprobar si uno de los operandos es mayor que `>`, menor que `<`, mayor o igual que `>=`, y menor o igual que `<=`. Estos operadores funcionan en cualquier tipo de operando, pero tienen sentido usarlos solo en números o valores que se convertirán correctamente en números.

```
console.log(10 > 100); // -> false
console.log(101 > 100); // -> true
console.log(101 > "100"); // -> true

console.log(101 < 100); // -> false
console.log(100n < 102); // -> true
console.log("10" < 20n); // -> true

console.log(101 <= 100); // -> false
console.log(10 >= 10n); // -> true
console.log("10" <= 20); // -> true
```

```
1 console.log("b" > "a"); // -> true
2 console.log("a" > "B"); // -> true
3 console.log("B" > "a"); // -> true
4 console.log("a" > "4"); // -> true
5 console.log("4" > "1"); // -> true
6
7 console.log("ab1" < "ab4"); // -> true
8 console.log("ab4" < "abA"); // -> true
9 console.log("abB" < "abA"); // -> true
10 console.log("aba" < "abb"); // -> true
11
12 console.log("ab" < "ab4"); // -> true
13
14
15
```

app.js

Console>\_
  
true
  
true
  
true
  
true
  
true
  
true
  
true
  
true

[Fullscreen](#)

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). [X](#)

## Modulo 3

### Sección 3

[Sandbox](#)

## Cuadros de diálogo

Los cuadros de diálogo son partes integrales de los navegadores web y están disponibles en casi todos ellos, incluso en los más antiguos. Todos ellos son ventanas emergentes (o ventanas modales), lo que significa que cuando se muestra el cuadro de diálogo, no es posible interactuar con la página web en sí hasta que se cierra este cuadro de diálogo.

Este inconveniente cuando la ventana emergente está visible es una de las razones por las que no debemos abusar de ellos. Están perfectamente bien para el proceso de aprendizaje, y en algunos casos excepcionales en los que se debe mostrar información importante o es obligatoria alguna entrada del

```
1 alert("¡Hola, Mundo!")
2 window.alert("¡Hola, Mundo!, por segunda ocasión");
3
4 alert(4 * 7);
5 alert(true);
6
7 alert("texto 1", "texto 2"); // solo "texto 1" será mostrado
8
```

**Cuadros de diálogo**

Los cuadros de diálogo son partes integrales de los navegadores web y están disponibles en casi todos ellos, incluso en los más antiguos. Todos ellos son ventanas emergentes (o ventanas modales), lo que significa que cuando se muestra el cuadro de diálogo, no es posible interactuar con la página web en sí hasta que se cierra este cuadro de diálogo.

Este inconveniente cuando la ventana emergente está visible es una de las razones por las que no debemos abusar de ellos. Están perfectamente bien para el proceso de aprendizaje, y en algunos casos excepcionales en los que se debe mostrar información importante o es obligatoria alguna entrada del usuario, pero se deben evitar en otras circunstancias. Tenemos tres cuadros de diálogo disponibles para usar.

**Cuadro de diálogo de alerta**

El cuadro de diálogo de alerta es el más simple de los tres, y para mostrar un cuadro de alerta, necesitamos llamar a un método llamado `alert()`. El método `alert` acepta un parámetro opcional: el texto que se mostrará. El método `alert` es un método de la ventana de objetos, pero por conveniencia, se puede usar sin necesidad de escribir `window.alert`, por

```
4 alert(4 * 7);
5 alert(true);
6
7 alert("texto 1", "texto 2"); // solo "texto 1" será mostrado
8
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**Cuadros de diálogo**

Los cuadros de diálogo son partes integrales de los navegadores web y están disponibles en casi todos ellos, incluso en los más antiguos. Todos ellos son ventanas emergentes (o ventanas modales), lo que significa que cuando se muestra el cuadro de diálogo, no es posible interactuar con la página web en sí hasta que se cierra este cuadro de diálogo.

Este inconveniente cuando la ventana emergente está visible es una de las razones por las que no debemos abusar de ellos. Están perfectamente bien para el proceso de aprendizaje, y en algunos casos excepcionales en los que

```
4 alert(4 * 7);
5 alert(true);
6
7 alert("texto 1", "texto 2"); // solo "texto 1" será mostrado
8
```

**Cuadros de diálogo**

Los cuadros de diálogo son partes integrales de los navegadores web y están disponibles en casi todos ellos, incluso en los más antiguos. Todos ellos son ventanas emergentes (o ventanas modales), lo que significa que cuando se muestra el cuadro de diálogo, no es posible interactuar con la página web en

```
4 alert(4 * 7);
5 alert(true);
6
7 alert("texto 1", "texto 2"); // solo "texto 1" será mostrado
8
```

## Cuadros de diálogo de confirmación

El segundo cuadro de diálogo se denomina cuadro de diálogo `confirm`. Al igual que `alert`, es un método que acepta un parámetro opcional: un mensaje que se mostrará. La diferencia entre `alert` y `confirm` es que el cuadro de diálogo `confirm` muestra dos botones, el botón Aceptar y el botón Cancelar. Dependiendo del botón presionado por el usuario, el método `confirm` devuelve un valor booleano. Se devuelve verdadero cuando el usuario cierra el cuadro de diálogo con el botón Aceptar y se devuelve falso cuando el usuario presiona el botón Cancelar.

```
let decision = window.confirm("¿Está bien?");
console.log(decision);
```

Los valores verdadero o falso, devueltos por el método `confirm` como resultado de la decisión del usuario, permiten la ejecución condicional de algunas acciones del programa. En el siguiente ejemplo, también hemos utilizado un operador condicional aprendido recientemente:

```
let remove = confirm("¿Eliminar todos los datos?");
let message = remove ? "Eliminando Datos" : "Cancelado"
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**Aceptar**

**Cancelar**

## Cuadros de diálogo de confirmación

El segundo cuadro de diálogo se denomina cuadro de diálogo `confirm`. Al igual que `alert`, es un método que acepta un parámetro opcional: un mensaje que se mostrará. La diferencia entre `alert` y `confirm` es que el cuadro de diálogo `confirm` muestra dos botones, el botón Aceptar y el botón Cancelar. Dependiendo del botón presionado por el usuario, el método `confirm` devuelve un valor booleano. Se devuelve verdadero cuando el usuario cierra el cuadro de diálogo con el botón Aceptar y se devuelve falso cuando el usuario presiona el botón Cancelar.

```
let decision = window.confirm("¿Está bien?");
console.log(decision);
```

Los valores verdadero o falso, devueltos por el método `confirm` como resultado de la decisión del usuario, permiten la ejecución condicional de algunas acciones del programa. En el siguiente ejemplo, también hemos utilizado un operador condicional aprendido recientemente:

```
let remove = confirm("¿Eliminar todos los datos?");
let message = remove ? "Eliminando Datos" : "Cancelado"
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

## Cuadros de diálogo de aviso (prompt)

El último de los cuadros de diálogo es el cuadro de diálogo `prompt`. Es un desarrollo posterior de la ventana emergente `confirm`. Al igual que el cuadro de diálogo `confirm`, contiene los botones Aceptar y Cancelar, pero también contiene un campo de texto de una sola línea que permite al usuario ingresar texto.

Al igual que con otros cuadros de diálogo, el `prompt` acepta un parámetro opcional como un mensaje que se mostrará. El `prompt` también acepta un segundo parámetro opcional, que es el valor predeterminado del campo de texto visible en la ventana de diálogo. Al igual que `confirm`, el método `prompt` devolverá un resultado que depende de la entrada del usuario.

Si el usuario cierra la ventana con el botón Aceptar, todo lo que se encuentre en el campo de texto se devolverá desde el método `prompt` como una cadena. Si el cuadro de diálogo se cierra con el botón Cancelar, el método `prompt` devolverá un valor null. Debido a que al pulsar el botón Aceptar el valor devuelto será de tipo String, en ocasiones necesitaremos convertirlo a otro tipo, por ejemplo, a un tipo Number. Al igual que con todas las entradas de los usuarios, debemos estar preparados para el hecho de que los datos

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

The screenshot shows a web page from JS INSTITUTE with a modal dialog box. The dialog box has a title "edube.org dice" and asks "¿Cuál es tu nombre?". A text input field contains "Brayan". Below the input are two buttons: "Aceptar" (Accept) and "Cancelar" (Cancel). In the background, the main content area has some text about the prompt dialog. On the right side of the screen, there is a developer tools panel titled "app.js" which shows the following code:

```
tu nombre?", "Juan Pérez");  
"¿Cuántos años tienes?");  
"los");
```

The screenshot shows a web page from JS INSTITUTE with a modal dialog box. The dialog box has a title "edube.org dice" and asks "Hola Brayan ¿Cuántos años tienes?". A text input field contains "20". Below the input are two buttons: "Aceptar" (Accept) and "Cancelar" (Cancel). In the background, the main content area has some text about the prompt dialog. On the right side of the screen, there is a developer tools panel titled "app.js" which shows the following code:

```
tu nombre?", "Juan Pérez");  
"¿Cuántos años tienes?");  
"los");
```

The screenshot shows a web page from JS INSTITUTE with a modal dialog box. The dialog box has a title "edube.org dice" and displays the message "Brayan tiene 20 años". Below the message is a single button "Aceptar" (Accept). In the background, the main content area has some text about the prompt dialog. On the right side of the screen, there is a developer tools panel titled "app.js" which shows the following code:

```
4 let edad = prompt("Hola " + nombre + " ¿Cuántos años tienes?");  
5 alert(nombre + " tiene " + edad + " años");  
6 |
```



## Tareas

### Tarea 1

Usando todo lo que has aprendido hasta este punto, escribe una secuencia de comandos que le pregunte al usuario sobre el ancho, alto y largo de una caja, luego calcula y devuelve al usuario el volumen de esta caja.

Como ejemplo, una caja con `anchura = 20`, `altura = 10` y `longitud = 50` tendrá un volumen de 10000 (omitiendo unidades, ya que no son relevantes en este escenario). Por ahora, supón que los valores proporcionados por el usuario son números válidos, pero si tienes alguna idea sobre cómo hacerlo, puedes intentar hacer este script de tal manera que sea resistente a los valores no válidos.

[Ejemplo](#)

```
1 // Solicitar al usuario el ancho de la caja
2 let ancho = parseFloat(prompt("Ingrese el ancho de la caja:"));
3
4 // Solicitar al usuario la altura de la caja
5 let altura = parseFloat(prompt("Ingrese la altura de la caja:"));
6
7 // Solicitar al usuario la longitud de la caja
8 let longitud = parseFloat(prompt("Ingrese la longitud de la caja:"));
9
10 // Calcular el volumen de la caja
11 let volumen = ancho * altura * longitud;
12
13 // Mostrar el resultado al usuario
14 alert("El volumen de la caja es: " + volumen);
15
```

**app.js**

**Console>...**



We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Tareas

### Tarea 1

Usando todo lo que has aprendido hasta este punto, escribe una secuencia de comandos que le pregunte al usuario sobre el ancho, alto y largo de una caja, luego calcula y devuelve al usuario el volumen de esta caja.

Como ejemplo, una caja con `anchura = 20`, `altura = 10` y `longitud = 50` tendrá un volumen de 10000 (omitiendo unidades, ya que no son relevantes en este escenario). Por ahora, supón que los valores proporcionados por el usuario son números válidos, pero si tienes alguna idea sobre cómo hacerlo, puedes intentar hacer este script de tal manera que sea resistente a los valores no válidos.

[Ejemplo](#)

edube.org dice

Ingrese la longitud de la caja:

**Aceptar** **Cancelar**

```
7 // Solicitar al usuario la longitud de la caja
8 let longitud = parseFloat(prompt("Ingrese la longitud de la caja:"));
9
10 // Calcular el volumen de la caja
11 let volumen = ancho * altura * longitud;
12
13 // Mostrar el resultado al usuario
14 alert("El volumen de la caja es: " + volumen);
15
```

**app.js**

**Console>...**



We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Tareas

### Tarea 1

Usando todo lo que has aprendido hasta este punto, escribe una secuencia de comandos que le pregunte al usuario sobre el ancho, alto y largo de una caja, luego calcula y devuelve al usuario el volumen de esta caja.

Como ejemplo, una caja con `anchura = 20`, `altura = 10` y `longitud = 50` tendrá un volumen de 10000 (omitiendo unidades, ya que no son relevantes en este escenario). Por ahora, supón que los valores proporcionados por el usuario son números válidos, pero si tienes alguna idea sobre cómo hacerlo, puedes intentar hacer este script de tal manera que sea resistente a los valores no válidos.

[Ejemplo](#)

edube.org dice

El volumen de la caja es: NaN

**Aceptar**

```
4 // Solicitar al usuario la altura de la caja
5 let altura = parseFloat(prompt("Ingrese la altura de la caja:"));
6
7 // Solicitar al usuario la longitud de la caja
8 let longitud = parseFloat(prompt("Ingrese la longitud de la caja:"));
9
10 // Calcular el volumen de la caja
11 let volumen = ancho * altura * longitud;
12
13 // Mostrar el resultado al usuario
14 alert("El volumen de la caja es: " + volumen);
15
```

**app.js**

**Console>...**



We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

**LABORATORIO**

**Tiempo estimado**  
15-30 minutos

**Nivel de dificultad**  
Fácil

**Objetivos**  
Familiarizar al alumno con:

- La interacción con el usuario (cuadros de diálogo: alerta, confirmación y aviso)

**Escenario**

Volvamos a nuestra lista de contactos. Después de algunos ajustes recientes (es decir, el emplear un arreglo y objetos), en realidad comienza a parecerse a una lista. Sin embargo, quedaba un problema bastante grave. El cambio de datos en la lista, como agregar un nuevo contacto, debe proporcionarse de

```

1 let contacts = [
2   name: "Maxwell Wright",
3   phone: "(0191) 719 6495",
4   email: "egestas@nonummyac.co.uk"
5   ],
6   name: "Baja Villarreal",
7   phone: "0866 398 2895",
8   email: "posuere@sed.com"
9   ],
10  name: "Helen Richards",
11  phone: "0800 1111",
12  email: "libero@vallis.edu"
13 ];
14 // escribe tu código aquí
15

```

app.js

Console >...

Maxwell Wright / (0191) 719 6495 /  
egestas@nonummyac.co.uk  
Helen Richards / 0800 1111 / libero@vallis.edu

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

## Modulo 4

### Sección

**Ejecución condicional**

La ejecución condicional o las sentencias de control de flujo ya se han mencionado varias veces y ahora es el momento de examinarlas con detalle. Este es un tema muy importante, ya que las sentencias control de flujo son esenciales no solo para JavaScript, sino también para la programación en general. Sin la capacidad de reaccionar y cambiar su comportamiento, cualquier código siempre haría lo mismo. Por supuesto, esto es a veces exactamente lo que necesitamos, pero la mayoría de las veces necesitamos capacidad de respuesta y la capacidad de manejar diferentes situaciones en el código.

Podemos imaginar nuestro programa como un árbol que comienza desde el tronco y se divide gradualmente en ramas. El tronco es el comienzo del programa, y cada instrucción condicional es un reflejo de una nueva rama. Las instrucciones condicionales se ejecutan sobre la base de la decisión del usuario, los resultados de cálculos anteriores u otra información que el programa tendrá en cuenta. JavaScript proporciona algunas formas de bifurcar la ejecución del código, basado en condiciones arbitrarias. A partir de este capítulo, habrá más tareas y código que deberás escribir, ya que ahora deberías tener a mano casi todas las herramientas necesarias.

```

1 let isUserReady = confirm("¿Estás listo?");
2 console.log(isUserReady);
3 if (isUserReady) {
4   alert("¡Usuario listo!");
5 }
6

```

app.js

Console >...

true

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

The screenshot shows a web-based JavaScript development environment. At the top, there's a header with the JS INSTITUTE logo and a 'Sandbox' button. A modal dialog box is open in the center, displaying the text "edube.org dice" and the question "¿Estás listo?". Below the modal is a code editor window titled "app.js" containing the following code:

```
1 if(confirm("¿Estás listo?")) {
2     console.log("Usuario listo!");
3     if(isUserReady) {
4         alert("¡Usuario listo!");
5     }
6 }
```

At the bottom of the code editor, there's a "Console" tab showing the output "true". On the right side of the interface, there are "Fullscreen" and "Sandbox" buttons.

This screenshot is similar to the one above, showing the same interface and code editor. The modal dialog now displays the message "¡Usuario listo!". The code in the "app.js" editor remains the same:

```
1 if(confirm("¿Estás listo?")) {
2     console.log("Usuario listo!");
3     if(isUserReady) {
4         alert("¡Usuario listo!");
5     }
6 }
```

This screenshot shows the state after the user has interacted with the application. The modal dialog is still present with the message "¡Usuario listo!". The code in the "app.js" editor has been modified to include a call to "alert" instead of "console.log".

```
1 let isUserReady = confirm("¿Estás listo?");
2 console.log(isUserReady);
3 if (isUserReady) {
4     alert("¡Usuario listo!");
5 }
```

This screenshot shows the final state of the application. The modal dialog is still visible with the message "¡Usuario listo!". The code in the "app.js" editor has been updated to use both "console.log" and "alert" statements.

```
1 let isUserReady = confirm("¿Estás listo?");
2 if (isUserReady)
3     console.log("¡Usuario listo!");
4     alert("¡Usuario listo!");
```

Below the code editor, a note says: "Corrige este código cerrando ambos comandos (console.log y alert) en el bloque. Comprueba cómo afectará esto al programa." The "Console" tab at the bottom shows the output "true".

**JS INSTITUTE**  
Open Education & Development Group

**La instrucción if**

En el ejemplo, hay una línea dentro del bloque de código if, lo que significa que podríamos omitir las llaves alrededor del bloque. Sin embargo, aunque puede parecer tentador hacerlo, siempre es mejor usar llaves. De esa forma, el código es más limpio y fácil de leer, y también evita un error muy común que ocurre cuando se intenta agregar otra instrucción dentro de un bloque if y se olvida agregar las llaves.

En el siguiente ejemplo, probablemente nos olvidamos de cerrar los dos comandos dentro del bloque directamente detrás de la instrucción if. Comprueba cómo se comportará este fragmento de código cuando lo ejecutes, según tu respuesta a la pregunta "¿Estás listo?".

```
let isUserReady = confirm("¿Estás listo?");

if (isUserReady)
  console.log("¡Usuario listo!");
  alert("¡Usuario listo!");
```

Corrige este código cerrando ambos comandos (console.log y alert) en el bloque. Comprueba cómo afectará esto al programa.

edube.org dice  
¡Usuario listo!

Aceptar

```
4 alert("¡Usuario listo!");
5 }
```

app.js

Console >\_ Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**JS INSTITUTE**  
Open Education & Development Group

**La instrucción if**

En el ejemplo, hay una línea dentro del bloque de código if, lo que significa que podríamos omitir las llaves alrededor del bloque. Sin embargo, aunque puede parecer tentador hacerlo, siempre es mejor usar llaves. De esa forma, el código es más limpio y fácil de leer, y también evita un error muy común que ocurre cuando se intenta agregar otra instrucción dentro de un bloque if y se olvida agregar las llaves.

En el siguiente ejemplo, probablemente nos olvidamos de cerrar los dos comandos dentro del bloque directamente detrás de la instrucción if. Comprueba cómo se comportará este fragmento de código cuando lo ejecutes, según tu respuesta a la pregunta "¿Estás listo?".

```
let isUserReady = confirm("¿Estás listo?");

if (isUserReady)
  console.log("¡Usuario listo!");
  alert("¡Usuario listo!");
```

Corrige este código cerrando ambos comandos (console.log y alert) en el bloque. Comprueba cómo afectará esto al programa.

edube.org dice  
¿Estás listo?

Aceptar Cancelar

```
4 alert("¡Usuario listo!");
5 }
```

app.js

Console >\_ Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**JS INSTITUTE**  
Open Education & Development Group

**La instrucción if - continuación**

Este es un buen momento para recordarte las operaciones lógicas y de comparación, ya que se usan más comúnmente como condiciones, especialmente en situaciones más complejas. Veamos el ejemplo:

```
let userAge = 23;
let isFemale = false;
let points = 703;
let cartValue = 299;
let shippingCost = 9.99;

if (userAge > 21) {
  if (cartValue >= 300 || points >= 500) {
    shippingCost = 0;
  }
}

console.log(shippingCost);
```

En este ejemplo, para establecer la variable `shippingCost` a cero, la variable `userAge` necesita ser mayor que 21. Para entrar al segundo if, la

« 4.1.1.4 La instrucción if - continuación »

app.js

Console >\_ Fullscreen

 Sandbox

### La instrucción if ... else

La instrucción if es muy útil, pero ¿qué pasa si también queremos ejecutar algún código cuando no se cumple una condición dada? Por supuesto, podríamos usar otra declaración if y cambiar la condición, como se muestra en el ejemplo:

```
let isUserReady = confirm("¿Estás listo?");

if (isUserReady) {
    console.log("¡Usuario listo!");
}

if (isUserReady == false) {
    console.log("¡Usuario no está listo!");
}
```

Esto funcionará como se esperaba, pero no se ve muy bien. ¿Y si en el futuro tenemos que cambiar esta condición? ¿Recordaremos cambiarlo en ambos lugares? Este es un posible error lógico futuro. Entonces, ¿qué podemos hacer? Podemos usar una palabra clave `else`.

```
1 let isUserReady = confirm("¿Estás listo?");
2
3 if (isUserReady) {
4     console.log("¡Usuario listo!");
5 } else {
6     console.log("¡Usuario no está listo!");
7 }
8
9
```

 app.js

Console>\_
jUsuario listo!

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

¿Estás listo?

 Aceptar

 Cancelar

 Sandbox

### La instrucción if ... else

La instrucción if es muy útil, pero ¿qué pasa si también queremos ejecutar algún código cuando no se cumple una condición dada? Por supuesto, podríamos usar otra declaración if y cambiar la condición, como se muestra en el ejemplo:

```
let isUserReady = confirm("¿Estás listo?");

if (isUserReady) {
    console.log("¡Usuario listo!");
}

if (isUserReady == false) {
    console.log("¡Usuario no está listo!");
}
```

Esto funcionará como se esperaba, pero no se ve muy bien. ¿Y si en el futuro tenemos que cambiar esta condición? ¿Recordaremos cambiarlo en ambos lugares? Este es un posible error lógico futuro. Entonces, ¿qué podemos hacer? Podemos usar una palabra clave `else`.

```
1 let isUserReady = confirm("¿Estás listo?");
2
3 if (isUserReady) {
4     console.log("¡Usuario listo!");
5 } else {
6     console.log("¡Usuario no está listo!");
7 }
8
9
```

 app.js

Console>\_
jUsuario listo!

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 app.js

Console>\_

¡Usuario listo!

 Fullscreen

### La instrucción if ... else ... if

Las instrucciones `if` y `if... else` nos dan una gran flexibilidad en cómo se puede comportar el código en relación con cualquier otra cosa. Pero ramificar el flujo de ejecución del código solo en dos ramas a veces no es suficiente. Hay una solución simple para esto en JavaScript: podemos anidar instrucciones `if ... else`. ¿Cómo funciona esto? Un segmento `else` puede tener una sentencia `if` o `if... else` dentro de él, y es posible anidar cualquier número de sentencias `if... else` si es necesario.

La sintaxis para esto se ve así:

```
if (condición_1) {
    code
} else if (condición_2) {
    code
} else if (condición_3) {
    code
} else {
    code
}
```

```
3+ if (number < 10) {
4+     alert("<10");
5+ } else if (number < 30) {
6+     alert("<30");
7+ } else if (number < 60) {
8+     alert("<60");
9+ } else if (number < 90) {
10+    alert("<90");
11+ } else if (number < 100) {
12+    alert("<100");
13+ } else if (number == 100) {
14+     alert("100")
15+ } else {
16+     alert(">100")
17+
18|
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

Ingrésa un número

30

Aceptar Cancelar

### La instrucción if ... else ... if

Las instrucciones `if` y `if... else` nos dan una gran flexibilidad en cómo se puede comportar el código en relación con cualquier otra cosa. Pero ramificar el flujo de ejecución del código solo en dos ramas a veces no es suficiente. Hay una solución simple para esto en JavaScript: podemos anidar instrucciones `if ... else`. ¿Cómo funciona esto? Un segmento `else` puede tener una sentencia `if` o `if... else` dentro de él, y es posible anidar cualquier número de sentencias `if... else` si es necesario.

La sintaxis para esto se ve así:

```
if (condición_1) {
    code
} else if (condición_2) {
    code
} else if (condición_3) {
    code
} else {
    code
}
```

```
9+ } else if (number < 90) {
10+    alert("<90");
11+ } else if (number < 100) {
12+    alert("<100");
13+ } else if (number == 100) {
14+     alert("100")
15+ } else {
16+     alert(">100")
17+
18|
```

app.js

Console>...

 Fullscreen

<60

Aceptar

### La instrucción if ... else ... if

Las instrucciones `if` y `if... else` nos dan una gran flexibilidad en cómo se puede comportar el código en relación con cualquier otra cosa. Pero ramificar el flujo de ejecución del código solo en dos ramas a veces no es suficiente. Hay una solución simple para esto en JavaScript: podemos anidar instrucciones `if ... else`. ¿Cómo funciona esto? Un segmento `else` puede tener una sentencia `if` o `if... else` dentro de él, y es posible

```
5+ } else if (number < 30) {
6+     alert("<30");
7+ } else if (number < 60) {
8+     alert("<60");
9+ } else if (number < 90) {
10+    alert("<90");
11+ } else if (number < 100) {
12+    alert("<100");
13+ } else if (number == 100) {
14+     alert("100")
```

 Fullscreen

### La instrucción if ... else ... if - continuación

En el siguiente ejemplo, podemos ver el uso de ifs en cascada con else, y también condiciones lógicas complejas. Siéntete libre de jugar con los valores asignados a las variables para ver cómo cambian los resultados.

Para resumir lo que está pasando, podemos disecar cada caso por separado:

- Si `userAge` es mayor que 65 O
- Si `userAge` es mayor o igual que 21 y uno de los siguientes casos:
  - `hasParentsApproval` es verdadero.
  - `cartValue` es mayor que 300.
  - `points` es mayor que 500.
 entonces `shippingCost` se reducirá a cero.
- Sino y si `userAge` es menor que 21 y `hasParentsApproval` es verdadero, `shippingCost` se reducirá en 5.
- Si `userAge` es menor que 21 pero `hasParentsApproval` es falso, la orden es inválida.

```

1 const INSURANCE_COST = 2.99;
2
3 let shippingCost = 0.99;
4 let isOrderValid = true;
5
6 let userAge = 22;
7 let points = 400;
8 let cartValue = 199;
9 let hasPromoCode = false;
10 let hasParentsApproval = false;
11 let addInsurance = true;
12
13 /* calcular costos de envío*/
14 if ((userAge > 65) || (userAge >= 21 && (hasPromoCode || cartValue > 300 || points > 500)))
15   shippingCost = 0;
16 else if (userAge < 21 && hasParentsApproval) {
17   shippingCost -= 5;
18 }
19 else {
20   shippingCost = 0;
21 }
22
23 console.log(`Shipping cost: ${shippingCost}`);
24
25 if (!isOrderValid) {
26   console.log(`Order is invalid`);
27 }
28
29 
```

app.js   index.html   style.css

Console>...

12.98

✖ Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). ✖

### Operador condicional

Hemos hablado del operador condicional (ternario) en la parte del curso dedicada a los operadores. Nos permite realizar una de dos acciones en función del valor del primer operando. La mayoría de las veces se usa como una alternativa a la instrucción `if ... else` cuando deseas asignar uno de dos valores a una variable, dependiendo de una determinada condición. En tales casos, es simplemente más corto y un poco más legible que la instrucción `if...else`. Veamos un ejemplo simple, sin usar un operador condicional:

```

let price = 100;
let shippingCost;
if (price > 50) {
  shippingCost = 0;
} else {
  shippingCost = 5;
}
console.log(`price = ${price}, shipping = ${shippingCost}`)

```

El mismo código reescrito con el operador condicional parece un poco más sencillo. Como recordatorio: el primer operando (`notes`) del signo de

```

1 let price = 100;
2 let shippingCost;
3 if (price > 50) {
4   shippingCost = 0;
5 } else {
6   shippingCost = 5;
7 }
8 console.log(`price = ${price}, shipping = ${shippingCost}`); // -> price = 100, shipping = 0
9 
```

app.js

Console>...

price = 100, shipping = 0

✖ Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). ✖

### La sentencia switch ... case

El último tipo de sentencia que se utiliza para la ejecución de código condicional es una sentencia `switch`. Estamos hablando de esto ahora porque, entre otras cosas, en comparación con la instrucción `if`, no es una sentencia que se use con mucha frecuencia. Es algo similar a las sentencias `if...else` anidadas, pero en lugar de evaluar diferentes expresiones, `switch` evalúa una expresión condicional y luego intenta hacer coincidir su valor con uno de los casos dados. Esta es la sintaxis de la sentencia `switch`:

```

switch (expresión) {
  case primera_coincidencia:
    código
    break;
  case segunda_coincidencia:
    código
    break;
  default:
    código
}

```

Comienza con la palabra clave reservada `switch` seguida de la expresión `a`

```

1 let gate = prompt("Elegir la puerta: a, b, o c");
2 let win = false;
3
4 switch (gate) {
5   case "a":
6     alert("Puerta A: Vacía");
7     break;
8   case "b":
9     alert("Puerta B: Premio Mayor");
10    win = true;
11    break;
12   case "c":
13     alert("Puerta C: Vacía");
14     break;
15   default:
16     alert("No existe la Puerta " + String(gate));
17 }
18 
```

app.js

Console>...

✖ Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). ✖

**JS INSTITUTE**  
Open Education & Development Group

### La sentencia switch ... case

El último tipo de sentencia que se utiliza para la ejecución de código condicional es una sentencia switch. Estamos hablando de esto ahora porque, entre otras cosas, en comparación con la instrucción if, no es una sentencia que se use con mucha frecuencia. Es algo similar a las sentencias if... else anidadas, pero en lugar de evaluar diferentes expresiones, switch evalúa una expresión condicional y luego intenta hacer coincidir su valor con uno de los casos dados. Esta es la sintaxis de la sentencia switch:

```
switch (expresión) {
  case primera_coincidencia:
    código
    break;
  case segunda_coincidencia:
    código
    break;
  default:
    código
}
```

Comienza con la palabra clave reservada `switch` seguida de la expresión a

edube.org dice  
Elegir la puerta: a, b, o c  
c  
Aceptar Cancelar

```
7   break;
8   case "b":
9     alert("Puerta B: Premio Mayor");
10    win = true;
11    break;
12   case "c":
13     alert("Puerta C: Vacia");
14     break;
15   default:
16     alert("No existe la Puerta " + String(gate));
```

app.js

Console>\_

Sandbox

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

### La sentencia switch ... case

El último tipo de sentencia que se utiliza para la ejecución de código condicional es una sentencia switch. Estamos hablando de esto ahora porque, entre otras cosas, en comparación con la instrucción if, no es una sentencia que se use con mucha frecuencia. Es algo similar a las sentencias if... else anidadas, pero en lugar de evaluar diferentes expresiones, switch evalúa una expresión condicional y luego intenta hacer coincidir su valor con uno de los casos dados. Esta es la sintaxis de la sentencia switch:

```
switch (expresión) {
  case primera_coincidencia:
    código
    break;
  case segunda_coincidencia:
    código
    break;
  default:
    código
}
```

Comienza con la palabra clave reservada `switch` seguida de la expresión a

edube.org dice  
Puerta C: Vacia  
Aceptar

```
4+ switch (gate) {
5   case "a":
6     alert("Puerta A: Vacia");
7     break;
8   case "b":
9     alert("Puerta B: Premio Mayor");
10    win = true;
11    break;
12   case "c":
13     alert("Puerta C: Vacia");
14     break;
15   default:
16     alert("No existe la Puerta " + String(gate));
```

app.js

Console>\_

Sandbox

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

« 4.1.11 Tareas: Condicionales »

### Tareas

#### Tarea 1

Escribe un script que le pida al usuario que ingrese un número. Muestra el mensaje: "`Bingo!`" cuando el número sea mayor que 90 pero menor que 110, de lo contrario muestra el mensaje: "`Fallaste!`". Utiliza la sentencia `if`.

Ejemplo

```
1 // Solicitar al usuario que ingrese un número
2 let numero = parseFloat(prompt("Por favor, ingresa un número:"));
3
4 // Comprobar si el número está entre 90 y 110
5 if (numero > 90 && numero < 110) {
6   console.log(";Bingo!");
7 } else {
8   console.log(";Fallaste!");
9 }
```

app.js

Sandbox

**JS INSTITUTE**  
Open Education & Development Group

## Tareas

**Tarea 1**

Escribe un script que le pida al usuario que ingrese un número. Muestra el mensaje "¡Bingo!" cuando el numero sea mayor que 90 pero menor que 110, de lo contrario muestra el mensaje: "¡Fallaste!". Utiliza la sentencia `if`.

**Ejemplo**

```

    1 // Solicitar al usuario que ingrese un número
  2 let numero = parseFloat(prompt("Por favor, ingresa un número:"));
  3
  4 // Comprobar si el número está entre 90 y 110
  5 if (numero > 90 && numero < 110) {
  6   console.log("¡Bingo!");
  7 } else {
  8   console.log("¡Fallaste!");
  9 }
 10
  
```

app.js

**JS INSTITUTE**  
Open Education & Development Group

## 4.1.1.11 Tareas: Condicionales

## Tareas

**Tarea 1**

Escribe un script que le pida al usuario que ingrese un número. Muestra el mensaje "¡Bingo!" cuando el numero sea mayor que 90 pero menor que 110, de lo contrario muestra el mensaje: "¡Fallaste!". Utiliza la sentencia `if`.

**Ejemplo**

**Tarea 2**

Reescribe el código anterior, reemplazando el `if` con un operador condicional ternario.

**Ejemplo**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

## 4.1.1.11 Tareas: Condicionales

el mensaje "¡Bingo!" cuando el numero sea mayor que 90 pero menor que 110, de lo contrario muestra el mensaje: "¡Fallaste!". Utiliza la sentencia `if`.

**Ejemplo**

**Tarea 2**

Reescribe el código anterior, reemplazando el `if` con un operador condicional ternario.

**Ejemplo**

**Tarea 3**

Escribe una aplicación de calculadora simple. Solicite al usuario la siguiente entrada, uno por uno: dos números y un carácter que representa una operación matemática de "+", "-", "\*", "+", o "/". Si la entrada del usuario es válida, calcula el resultado y muéstralos al usuario. Si la entrada del usuario no es válida, muestra un mensaje que informa

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

edube.org dice

Por favor, ingresa un número:

99

Aceptar Cancelar

```
vor, ingresa un número:");
: 110) ? ";Bingo!" : ";Fallaste!";
```

Ejemplo

### Tarea 2

Reescribe el código anterior, reemplazando el `if` con un operador condicional ternario.

Ejemplo app.js

**JS INSTITUTE**  
Open Education & Development Group

« 4.1.11 Tareas: Condicionales »

edube.org dice

Por favor, ingresa un número:

99

Aceptar Cancelar

```
vor, ingresa un número:");
: 110) ? ";Bingo!" : ";Fallaste!";
```

Ejemplo

### Tarea 2

Reescribe el código anterior, reemplazando el `if` con un operador condicional ternario.

Ejemplo app.js

### Tarea 3

Escribe una aplicación de calculadora simple. Solicite al usuario la siguiente entrada, uno por uno: dos números y un carácter que representa una operación matemática de "+", "-", "\*", "+", o "/". Si la entrada del usuario es válida, calcula el resultado y muéstralo al usuario. Si la entrada del usuario no es válida, muestra un mensaje que informa

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

« 4.1.11 Tareas: Condicionales »

edube.org dice

Por favor, ingresa un número:

99

Aceptar Cancelar

```
vor, ingresa un número:");
: 110) ? ";Bingo!" : ";Fallaste!";
```

Ejemplo

### Tarea 3

Escribe una aplicación de calculadora simple. Solicite al usuario la siguiente entrada, uno por uno: dos números y un carácter que representa una operación matemática de "+", "-", "\*", "+", o "/". Si la entrada del usuario es válida, calcula el resultado y muéstralo al usuario. Si la entrada del usuario no es válida, muestra un mensaje que informa al usuario que se ha producido un error. Recuerda que el valor devuelto por la función `prompt` es del tipo cadena. Puedes usar el método `Number.isNaN` para verificar si se obtiene el número correcto después de la conversión. Por ejemplo, llamar a `Number.NaN(10)` devolverá `false`, mientras que `Number.NaN(NaN)` devolverá `true`.

Ejemplo app.js

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

**Ejemplo**

**Tarea 3**

Escribe una aplicación de calculadora simple. Solicite al usuario la siguiente entrada, uno por uno: dos números y un carácter que representa una operación matemática de "+", "-", "\*", o "/". Si la entrada del usuario es válida, calcula el resultado y muéstralos al usuario. Si la entrada del usuario no es válida, muestra un mensaje que informa al usuario que se ha producido un error. Recuerda que el valor devuelto por la función prompt es del tipo cadena. Puedes usar el método

**edube.org dice**  
Ingrese el primer número:  
75  
Aceptar Cancelar

```

7+ } else {
8  // Solicitar al usuario la operación matemática
9  let operacion = prompt("Ingrese la operación matemática (+, -, *, /):");
10
11 // Solicitar al usuario el segundo número
12 let numero2 = parseFloat(prompt("Ingrese el segundo número:"));
13
14 // Verificar si la entrada del usuario es un número válido
15 if (Number.isNaN(numero2)) {
16   console.log("Error: La entrada no es un número válido.");
    
```

**app.js**

**JS INSTITUTE**  
Open Education & Development Group

**Ejemplo**

**Tarea 3**

Escribe una aplicación de calculadora simple. Solicite al usuario la siguiente entrada, uno por uno: dos números y un carácter que representa una operación matemática de "+", "-", "\*", o "/". Si la entrada del usuario es válida, calcula el resultado y muéstralos al usuario. Si la entrada del usuario no es válida, muestra un mensaje que informa al usuario que se ha producido un error. Recuerda que el valor devuelto por la función prompt es del tipo cadena. Puedes usar el método

**edube.org dice**  
Ingrese la operación matemática (+, -, \*, /):  
+  
Aceptar Cancelar

```

7+ } else {
8  // Solicitar al usuario la operación matemática
9  let operacion = prompt("Ingrese la operación matemática (+, -, *, /):");
10
11 // Solicitar al usuario el segundo número
12 let numero2 = parseFloat(prompt("Ingrese el segundo número:"));
13
14 // Verificar si la entrada del usuario es un número válido
15 if (Number.isNaN(numero2)) {
16   console.log("Error: La entrada no es un número válido.");
    
```

**app.js**

**JS INSTITUTE**  
Open Education & Development Group

**Ejemplo**

**Tarea 3**

Escribe una aplicación de calculadora simple. Solicite al usuario la siguiente entrada, uno por uno: dos números y un carácter que representa una operación matemática de "+", "-", "\*", o "/". Si la entrada del usuario es válida, calcula el resultado y muéstralos al usuario. Si la entrada del usuario no es válida, muestra un mensaje que informa al usuario que se ha producido un error. Recuerda que el valor devuelto por la función prompt es del tipo cadena. Puedes usar el método

**edube.org dice**  
Ingrese el segundo número:  
22  
Aceptar Cancelar

```

7+ } else {
8  // Solicitar al usuario la operación matemática
9  let operacion = prompt("Ingrese la operación matemática (+, -, *, /):");
10
11 // Solicitar al usuario el segundo número
12 let numero2 = parseFloat(prompt("Ingrese el segundo número:"));
13
14 // Verificar si la entrada del usuario es un número válido
15 if (Number.isNaN(numero2)) {
16   console.log("Error: La entrada no es un número válido.");
    
```

**app.js**

**JS INSTITUTE**  
Open Education & Development Group

**Ejemplo**

**Tarea 3**

Escribe una aplicación de calculadora simple. Solicite al usuario la siguiente entrada, uno por uno: dos números y un carácter que representa una operación matemática de "+", "-", "\*", o "/". Si la entrada del usuario es válida, calcula el resultado y muéstralos al usuario. Si la entrada del usuario no es válida, muestra un mensaje que informa al usuario que se ha producido un error. Recuerda que el valor devuelto por la función prompt es del tipo cadena. Puedes usar el método `Number.isNaN` para verificar si se obtiene el número correcto después de la conversión. Por ejemplo, llamar a `Number.isNaN(10)` devolverá `false`, mientras que `Number.isNaN(NaN)` devolverá `true`.

**edube.org dice**  
El resultado es: 1650  
Fullscreen

**Console**  
El resultado es: 1650

**app.js**

**LABORATORIO**

### Tiempo Estimado

15-30 minutos

### Nivel de Dificultad

Fácil

### Objetivos

Familiarizar al estudiante con:

- La ejecución condicional (qué es la ejecución condicional, la sentencia if-else, el operador condicional, la sentencia switch-case)

### Escenario

Durante la última modificación del programa con la lista de contactos, permitimos que el usuario agregara un nuevo contacto con los datos ingresados por el usuario mientras se ejecuta el programa. Vayamos un paso más allá: intenta modificar el programa para que el usuario pueda elegir lo

```
32         return foundContacts;
33     }
34
35 // Ejemplo de uso: buscar contactos por nombre, número de teléfono o correo electrónico
36 let query = prompt("Ingrese el nombre, número de teléfono o correo electrónico del contacto");
37 let results = searchContact(query);
38
39+ if (results.length > 0) {
40     console.log("Se encontraron los siguientes contactos:");
41+     results.forEach(contact => {
42         console.log(contact.name + " - Teléfono: " + contact.phone + " - Correo electrónico: " + contact.email);
43     });
44+ } else {
45     console.log("No se encontraron contactos que coincidan con la búsqueda.");
46 }
47
```

app.js

Console >...

No se encontraron contactos que coincidan con la búsqueda.

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**LABORATORIO**

### Tiempo Estimado

15-30 minutos

### Nivel de Dificultad

Fácil

### Objetivos

Familiarizar al estudiante con:

- La ejecución condicional (qué es la ejecución condicional, la sentencia if-else, el operador condicional, la sentencia switch-case)

### Escenario

Durante la última modificación del programa con la lista de contactos, permitimos que el usuario agregara un nuevo contacto con los datos ingresados por el usuario mientras se ejecuta el programa. Vayamos un paso más allá: intenta modificar el programa para que el usuario pueda elegir lo

edube.org dice

Ingrese el nombre, número de teléfono o correo electrónico del contacto que desea buscar:

Aceptar Cancelar

```
8        email: "posuere@sed.com"
9    },
10   name: "Helen Richards",
11   phone: "0800 1111",
12   email: "libero@convallis.edu"
13 ];
14
15+ function searchContact(query) {
16     let foundContacts = [];
17
18     contacts.forEach(contact => {
19         if (contact.name.toLowerCase().includes(query.toLowerCase()) ||
20             contact.phone.includes(query) ||
21             contact.email.includes(query)) {
22             foundContacts.push(contact);
23         }
24     });
25
26     return foundContacts;
27 }
```

app.js

Console >...

Uncaught TypeError: Cannot read properties of null (reading 'toLowerCase')

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**LABORATORIO**

### Tiempo Estimado

15-30 minutos

### Nivel de Dificultad

Fácil

### Objetivos

Familiarizar al estudiante con:

- La ejecución condicional (qué es la ejecución condicional, la sentencia if-else, el operador condicional, la sentencia switch-case)

### Escenario

Durante la última modificación del programa con la lista de contactos, permitimos que el usuario agregara un nuevo contacto con los datos ingresados por el usuario mientras se ejecuta el programa. Vayamos un paso más allá: intenta modificar el programa para que el usuario pueda elegir lo

```
1+ let contacts = [
2     {
3         name: "Maxwell Wright",
4         phone: "(0191) 719 6495",
5         email: "Curabitur@nconumyac.co.uk"
6     },
7     {
8         name: "Raja Villarreal",
9         phone: "0866 398 2895",
10        email: "posuere@sed.com"
11    },
12    {
13        name: "Helen Richards",
14        phone: "0800 1111",
15        email: "libero@convallis.edu"
16    }];
17
18+ function searchContact(query) {
19     let foundContacts = [];
20
21     contacts.forEach(contact => {
22         if (contact.name.toLowerCase().includes(query.toLowerCase()) ||
23             contact.phone.includes(query) ||
24             contact.email.includes(query)) {
25             foundContacts.push(contact);
26         }
27     });
28
29     return foundContacts;
30 }
```

app.js

Console >...

Se encontraron los siguientes contactos:  
Raja Villarreal - Teléfono: 0866 398 2895 - Correo electrónico: posuere@sed.com

Fullscreen

## Modulo 4

### Sección 2

**JS INSTITUTE**  
Open Education & Development Group

#### « 4.2.1.3 El bucle while »

**El bucle while - continuación**

El bucle `while` es tan versátil que alguien lo suficientemente persistente podría reemplazar todas las demás sentencias de control de flujo con bucles `while`, incluso sentencias `if`. Por supuesto, sería engoroso en el mejor de los casos. El bucle `while` es uno de los bucles que normalmente usamos cuando no sabemos exactamente cuántas veces se debe repetir algo, pero sabemos cuándo parar. La sintaxis del bucle `while` es la siguiente:

```
while(condición) {
    bloque de código
}
```

La expresión entre paréntesis se evalúa al comienzo de cada iteración del bucle. Si la condición se evalúa como verdadera, se ejecutará el código entre llaves. Luego, la ejecución vuelve al comienzo del bucle y la condición se evalúa nuevamente. El bucle iterará y el código se ejecutará siempre que la condición se evalúe como verdadera. Esto significa, por supuesto, que con una condición mal formada, un bucle `while` puede convertirse en un bucle infinito, un bucle sin final. Dependiendo del contexto, eso puede ser lo que queremos lograr. Casi todos los juegos de computadora, por ejemplo, son bucles `while`.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Sandbox**

```
1 let isOver = false;
2 let counter = 1;
3
4 while (!isOver) {
5     isOver = !confirm(`[ ${counter++}] ¿Continuar en el bucle?`);
6 }
7
```

**app.js**

**Console >...**

**Fullscreen**

**JS INSTITUTE**  
Open Education & Development Group

**edube.org dice**

[1] ¿Continuar en el bucle?

**Aceptar** **Cancelar**

**Sandbox**

```
4 while (!isOver) {
5     isOver = !confirm(`[ ${counter++}] ¿Continuar en el bucle?`);
6 }
7
```

**app.js**

**Console >...**

**Fullscreen**

**JS INSTITUTE**  
Open Education & Development Group

#### « 4.2.1.4 El bucle do ... while »

**El bucle do ... while**

El bucle `do ... while` es muy similar al bucle simple `while`, la principal diferencia es que en un bucle `while`, la condición se verifica antes de cada iteración, y en el bucle `do ... while`, la condición se verifica después de cada iteración. Esto no parece una gran diferencia, pero la consecuencia de esto es que un código de bucle `do ... while` siempre se ejecuta al menos una vez antes de que se realice la primera verificación de condición, y un `do`...`while` nunca se puede ejecutar si la condición inicial se evalúa como falsa al comienzo del bucle. La sintaxis del bucle `do ... while` es la siguiente:

```
do {
    bloque de código
} while(condición);
```

Reescribamos nuestro último ejemplo usando `do ... while` en lugar de `while`. Nota que esta vez la variable `isOver` no necesita inicializarse antes del bucle (la condición se verifica al final del bucle y se llamará al cuadro de diálogo de confirmación antes de la primera prueba).

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Sandbox**

```
1 let condition = false;
2
3 while (condition) {
4     console.log("Una iteración del bucle while."); // nunca se ejecuta
5 }
6
7 do {
8     console.log("Una iteración del bucle do ... while."); // ejecutado una vez
9 } while (condition);
10
11
```

**app.js**

**Console >...**

Una iteración del bucle do ... while.

**Fullscreen**

## « 4.2.1.6 El bucle for -continuación »

[Sandbox](#)

### El bucle for - continuación

En ambos casos (bucle for y bucle while), declaramos e iniciamos la variable i antes de que comience el bucle (establecida inicialmente en 0). Ambos bucles se ejecutarán siempre que la condición `i < 10` se cumpla. En ambos bucles, al final de cada iteración, el valor de la variable i se incrementará en 1. Y por supuesto, en ambos bucles de cada iteración, el valor actual de la variable i se imprimirá en la consola. Entonces puedes ver que el ciclo for en realidad ofrece una forma ligeramente diferente y más consistente de escribir lo mismo que se puede lograr con el ciclo while. Tal notación es particularmente útil en casos como el presentado en el ejemplo, donde usamos un contador de iteraciones (en nuestro ejemplo, la variable i), que debe ser inicializada e incrementada (o decrementada). En tal situación, todo lo relacionado con el contador (inicialización, condición de finalización del bucle, cambio de valor del contador) se encuentra prácticamente en un solo lugar, lo que puede aumentar la legibilidad del código.

Un ejemplo típico del uso de un bucle for, en el que sabemos el número de iteraciones por adelantado, es cuando se revisan los elementos de un arreglo. Pasemos a otro ejemplo sencillo. Supongamos que tenemos un arreglo de números enteros de cuatro elementos a nuestra disposición y nuestro sueño es sumar todos estos números. No hay nada más fácil que

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). [X](#)

app.js

Console&gt;\_

190

[Fullscreen](#)

## « 4.2.1.7 Bucles y arreglos »

[Sandbox](#)

### Bucles y arreglos

Intentemos jugar de nuevo con los arreglos. Esta vez el programa será un poco más complicado. Queremos que el usuario ingrese nombres durante la ejecución del programa (usaremos el cuadro de diálogo `prompt`, que se colocarán en el arreglo uno por uno. La introducción de nombres finalizará cuando se pulse el botón Cancelar. Luego, escribiremos todo el contenido del arreglo (es decir, todos los nombres ingresados) en la consola.

```
let names = [];
let isOver = false;
while (!isOver) {
  let name = prompt("Ingresa otro nombre o presiona cancelar");
  if (name != null) {
    names.push(name);
  } else {
    isOver = true;
  }
}

for (let i = 0; i < names.length; i++) {
  console.log(names[i]);
}
```

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). [X](#)

app.js

Console&gt;\_

10

30

50

100

100

50

30

10

[Fullscreen](#)

## « 4.2.1.8 for ... of »

[Sandbox](#)

### for ... of

Además del bucle for regular, hay dos versiones específicas, una de las cuales, `for...of`, está dedicada para usar con arreglos (y otras estructuras iterativas, que sin embargo están más allá del alcance de este curso). En un bucle de este tipo, no especificamos explícitamente ninguna condición ni número de iteraciones, ya que se realiza exactamente tantas veces como elementos haya en el arreglo indicado.

Volvamos a nuestro ejemplo de sumar los números del arreglo de cuatro elementos. En este ejemplo, usamos un bucle for simple:

```
let values = [10, 30, 50, 100];
let sum = 0;
for (let i = 0; i < values.length; i++) {
  sum += values[i];
}
console.log(sum); // -> 190
```

La versión de este programa que usa el bucle `for ... of` se verá ligeramente diferente:

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). [X](#)

app.js

Console&gt;\_

São Paulo (20880000)

Mexico City (21340000)

Shanghai (23480000)

Delhi (25870000)

Tokyo (37260000)

[Fullscreen](#)

« 4.2.1.9 for ... in »

The screenshot shows a code editor with the following code:

```

for ... in
    También existe una versión del bucle for que nos permite recorrer campos de objetos. La sintaxis es for ... in. Itera a través de todos los campos del objeto indicado, colocando los nombres de estos campos (o claves) en la variable. En el ejemplo, usamos un objeto que contiene datos sobre un usuario:
    
```

```

let user = {
  name: "Calvin",
  surname: "Hart",
  age: 66,
  email: "CalvinMHart@teleworm.us"
};

for (let key in user) {
  console.log(key); // -> name, surname, age, email
}
    
```

En cada iteración del bucle, los nombres de los campos sucesivos del objeto de usuario se asignan a la variable `key`, es decir: nombre, apellido, edad, correo electrónico. Luego los escribimos en la consola. Pero, ¿y si queremos

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

« 4.2.1.10 Las instrucciones break y continue »

### Las instrucciones break y continue

La instrucción `break` se utiliza para terminar la ejecución de un bucle o un switch. En cada uno de estos contextos, cada vez que el motor de JavaScript encuentra una instrucción `break`, sale de todo el bucle o el switch y salta a la primera instrucción después de ese bucle o switch.

En el ejemplo, podemos ver un bucle infinito `while` del que se saldrá usando `break` después de que la variable `i` sea mayor o igual a 5. Tal uso de `break` es solo de importancia ilustrativa, porque tendría más sentido incluir esta condición directamente en la construcción del `while`.

```

let i = 0;
// Un bucle infinito
while (true){
  console.log(i);
  i++;
  if (i >= 5) {
    break;
  }
}
    
```

The screenshot shows a code editor with the following code:

```

1 let i = 0;
2 // Un bucle infinito
3 while (true){
4   console.log(i);
5   i++;
6   if (i >= 6) {
7     break;
8   }
9 }
10 alert(`Se salio del bucle con un break (${i}).`);
11
    
```

app.js

Console >...

0  
1  
2  
3  
4

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

edube.org dice

Se salió del bucle con un break (6).

Aceptar

### Las instrucciones break y continue

La instrucción `break` se utiliza para terminar la ejecución de un bucle o un switch. En cada uno de estos contextos, cada vez que el motor de JavaScript encuentra una instrucción `break`, sale de todo el bucle o el switch y salta a la primera instrucción después de ese bucle o switch.

En el ejemplo, podemos ver un bucle infinito `while` del que se saldrá usando `break` después de que la variable `i` sea mayor o igual a 5. Tal uso de `break` es solo de importancia ilustrativa, porque tendría más sentido incluir esta condición directamente en la construcción del `while`.

```

let i = 0;
// Un bucle infinito
while (true){
  console.log(i);
  i++;
  if (i >= 5) {
    break;
  }
}
    
```

The screenshot shows a code editor with the following code:

```

1 let i = 0;
2 // Un bucle infinito
3 while (true){
4   console.log(i);
5   i++;
6   if (i >= 6) {
7     break;
8   }
9 }
10 alert(`Se salio del bucle con un break (${i}).`);
11
    
```

app.js

Console >...

0  
1  
2  
3  
4

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

### Las instrucciones break y continue

La instrucción `break` se utiliza para terminar la ejecución de un bucle o un `switch`. En cada uno de estos contextos, cada vez que el motor de JavaScript encuentra una instrucción `break`, sale de todo el bucle o el `switch` y salta a la primera instrucción después de ese bucle o `switch`.

En el ejemplo, podemos ver un bucle infinito `while` del que se saldrá usando `break` después de que la variable `i` sea mayor o igual a 5. Tal uso de `break` es solo de importancia ilustrativa, porque tendría más sentido incluir esta condición directamente en la construcción del `while`.

```
let i = 0;
// Un bucle infinito
while (true){
    console.log(i);
    i++;
    if (i >= 5) {
        break;
    }
}
```

```
1 let i = 0;
2 // Un bucle infinito
3 while (true){
4     console.log(i);
5     i++;
6     if (i >= 6) {
7         break;
8     }
9 }
10 alert(`Se salio del bucle con un break ${i}.`);
11
12 |
```

app.js

Console >...

0  
1  
2  
3  
4  
5

Fullscreen

### La palabra clave break

También necesitamos decir algunas palabras más sobre la palabra clave `break`. En el ejemplo, la palabra clave `break` está presente en todos los casos excepto en el caso `default`. A diferencia de las sentencias `if`, las sentencias `switch` no ejecutan solo una rama, sino que ejecutan el código completo desde el primer caso que coincide hasta el final de la sentencia `switch`. Este comportamiento se llama pasar a través de y tiene algunos usos, pero la mayoría de las veces queremos ejecutar solo una rama, y por eso está presente la palabra clave `break`. Cuando un intérprete de JavaScript llega a un `break`, saltará fuera de la sentencia `switch` actual.

Para entender esto mejor, observa este ejemplo ligeramente modificado de un `switch`:

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
    case "a":
        alert("Puerta A: Vacia");
    case "b":
```

```
4 switch (gate) {
5     case "a":
6         alert("Puerta A: Vacia");
7     case "b":
8         alert("Puerta B: Premio Mayor");
9         win = true;
10    case "c":
11        alert("Puerta C: Vacia");
12    default:
13        alert("No existe la puerta " + String(gate));
14    }
15
16    if (win) {
17        alert("Ganador!");
18    }
19 |
```

app.js

Console >...

Puerta A: Vacia  
Puerta B: Premio Mayor  
Ganador!

Fullscreen



### La palabra clave break

También necesitamos decir algunas palabras más sobre la palabra clave `break`. En el ejemplo, la palabra clave `break` está presente en todos los casos excepto en el caso `default`. A diferencia de las sentencias `if`, las sentencias `switch` no ejecutan solo una rama, sino que ejecutan el código completo desde el primer caso que coincide hasta el final de la sentencia `switch`. Este comportamiento se llama pasar a través de y tiene algunos usos, pero la mayoría de las veces queremos ejecutar solo una rama, y por eso está presente la palabra clave `break`. Cuando un intérprete de JavaScript llega a un `break`, saltará fuera de la sentencia `switch` actual.

Para entender esto mejor, observa este ejemplo ligeramente modificado de un `switch`:

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
    case "a":
        alert("Puerta A: Vacia");
    case "b":
```

```
10     case "c":
11         alert("Puerta C: Vacia");
12     default:
13         alert("No existe la puerta " + String(gate));
14    }
15
16    if (win) {
17        alert("Ganador!");
18    }
19 |
```

app.js

Console >...

Puerta A: Vacia  
Puerta C: Vacia  
Ganador!

Fullscreen

**La palabra clave break**

También necesitamos decir algunas palabras más sobre la palabra clave `break`. En el ejemplo, la palabra clave `break` está presente en todos los casos excepto en el caso `default`. A diferencia de las sentencias `if`, las sentencias `switch` no ejecutan solo una rama, sino que ejecutan el código completo desde el primer caso que coincide hasta el final de la sentencia `switch`. Este comportamiento se llama pasar a través de y tiene algunos usos, pero la mayoría de las veces queremos ejecutar solo una rama, y por eso está presente la palabra clave `break`. Cuando un intérprete de JavaScript llega a un `break`, saltará fuera de la sentencia `switch` actual.

Para entender esto mejor, observa este ejemplo ligeramente modificado de un `switch`:

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
  case "a":
    alert("Puerta A: Vacía");
  case "b":
```

edube.org dice  
Puerta A: Vacía

Aceptar

app.js

Console>\_

Fullscreen

**4.2.11.12 La palabra clave break - continuación**

La palabra clave `break` - continuación

El `break` puede ser útil cuando más de un caso debe terminar exactamente con el mismo comportamiento.

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
  case "a":
  case "B":
  case 1:
  case "1":
    alert("Puerta A: Vacía");
    break;
  case "b":
  case "B":
  case 2:
  case "2":
    alert("Puerta B: Premio Mayor");
    win = true;
    break;
  case "3":
```

app.js

Console>\_

Fullscreen

**La palabra clave break - continuación**

El `break` puede ser útil cuando más de un caso debe terminar exactamente con el mismo comportamiento.

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
  case "a":
  case "A":
  case 1:
  case "1":
    alert("Puerta A: Vacía");
    break;
  case "b":
  case "B":
  case 2:
  case "2":
    alert("Puerta B: Premio Mayor");
    win = true;
    break;
  case "3":
```

edube.org dice  
Elige una puerta: a, b, o c

b

Aceptar Cancelar

app.js

Console>\_

Fullscreen

**JS INSTITUTE**  
Open Education & Development Group

**La palabra clave break - continuación**

El break puede ser útil cuando más de un caso debe terminar exactamente con el mismo comportamiento.

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
    case "A":
        case 1:
        case "1":
            alert("Puerta A: Vacía");
            break;
    case "B":
    case "B":
        case 2:
        case "2":
            alert("Puerta B: Premio Mayor");
            break;
}
```

**edube.org dice**  
Puerta B: Premio Mayor

Aceptar

**Sandbox**

**JS INSTITUTE**  
Open Education & Development Group

**La palabra clave break - continuación**

El break puede ser útil cuando más de un caso debe terminar exactamente con el mismo comportamiento.

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
    case "A":
        case 1:
        case "1":
            alert("Puerta A: Vacía");
            break;
    case "B":
    case "B":
        case 2:
        case "2":
            alert("Puerta B: Premio Mayor");
            win = true;
            break;
}
```

**edube.org dice**  
¡Ganador!

Aceptar

**Sandbox**

**JS INSTITUTE**  
Open Education & Development Group

**4.2.1.13 La palabra clave break - continuación »**

**La palabra clave break - continuación**

La última parte importante es que si se necesita un código más complejo dentro de un caso determinado, debemos colocar ese código en bloques de código separados rodeándolo adicionalmente con llaves. Esto aumentará la legibilidad del código y permitirá la declaración de variables solo en el alcance del caso dado.

```
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
    case "a": {
        let message = "Puerta A";
        console.log(message);
        break;
    }
    case "b": {
        let message = "Puerta B";
        console.log(message);
        break;
    }
    case "c": {
        let message = "Puerta C";
        console.log(message);
        break;
    }
    default:
        alert("No existe la puerta " + String(gate));
}

if (win) {
    alert("¡Ganador!");
}
```

**app.js**

Console>...

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**JS INSTITUTE**  
Open Education & Development Group

**La palabra clave break - continuación**

Elige una puerta: a, b, o c

Aceptar Cancelar

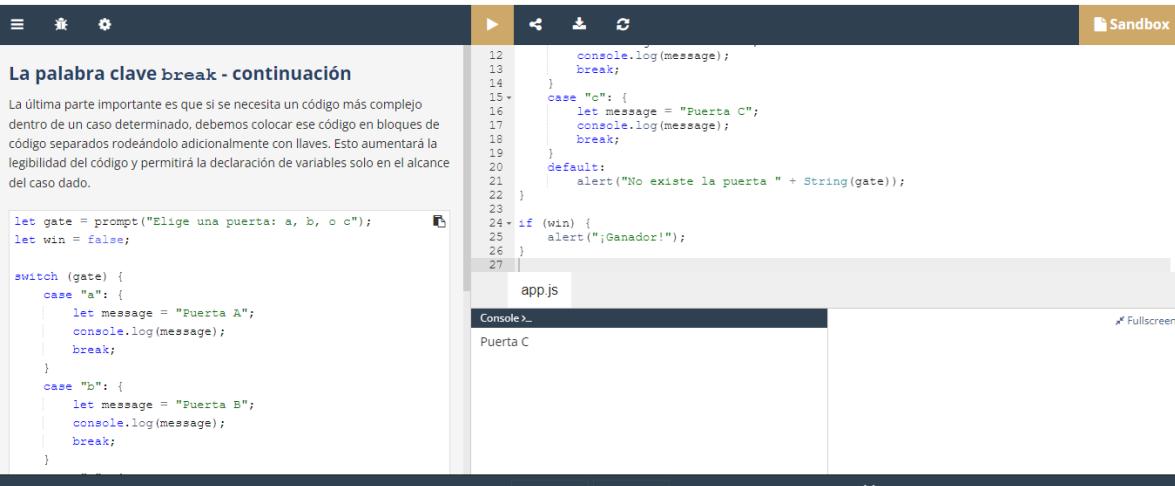
**edube.org dice**

```
let gate = prompt("Elige una puerta: a, b, o c");

switch (gate) {
    case "a": {
        let message = "Puerta A";
        console.log(message);
        break;
    }
    case "b": {
        let message = "Puerta B";
        console.log(message);
        break;
    }
    case "c": {
        let message = "Puerta C";
        console.log(message);
        break;
    }
    default:
        alert("No existe la puerta " + String(gate));
}

if (win) {
    alert("¡Ganador!");
}
```

**Sandbox**



**La palabra clave break - continuación**

La última parte importante es que si se necesita un código más complejo dentro de un caso determinado, debemos colocar ese código en bloques de código separados rodeándolo adicionalmente con llaves. Esto aumentará la legibilidad del código y permitirá la declaración de variables solo en el alcance del caso dado.

```

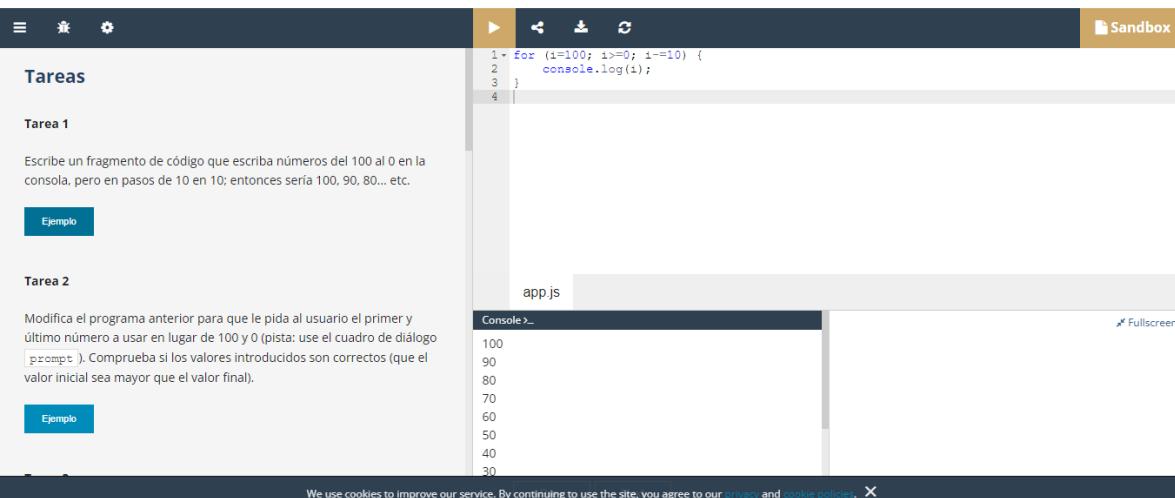
let gate = prompt("Elige una puerta: a, b, o c");
let win = false;

switch (gate) {
    case "a": {
        let message = "Puerta A";
        console.log(message);
        break;
    }
    case "b": {
        let message = "Puerta B";
        console.log(message);
        break;
    }
    case "c": {
        let message = "Puerta C";
        console.log(message);
        break;
    }
    default:
        alert("No existe la puerta " + String(gate));
}
if (win) {
    alert(";Ganador!");
}

```

app.js

Console>... Puerta C



**Tareas**

**Tarea 1**

Escribe un fragmento de código que escriba números del 100 al 0 en la consola, pero en pasos de 10 en 10: entonces sería 100, 90, 80... etc.

**Ejemplo**

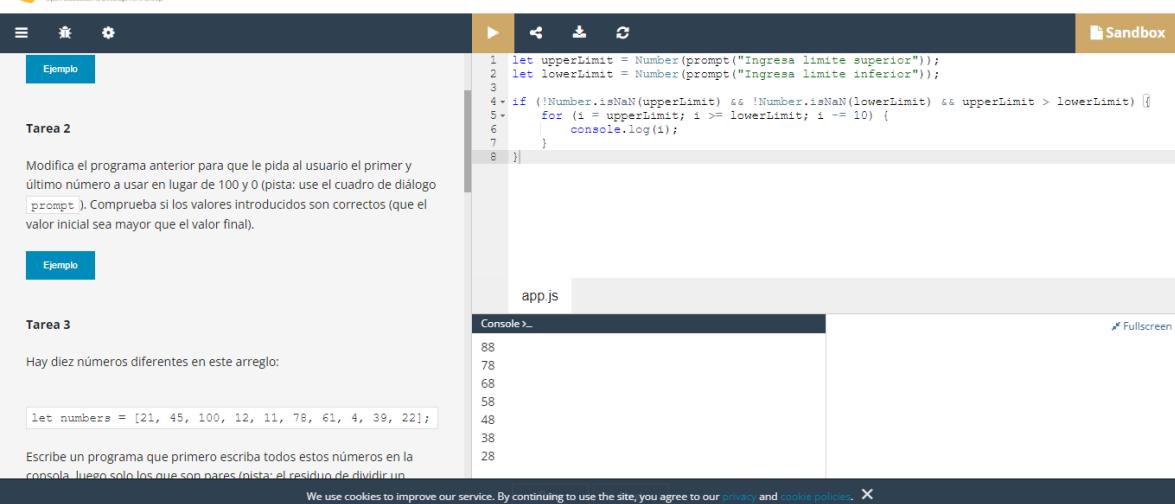
```

for (i=100; i>=0; i-=10) {
    console.log(i);
}

```

app.js

Console>... 100  
90  
80  
70  
60  
50  
40  
30



**Ejemplo**

**Tarea 2**

Modifica el programa anterior para que le pida al usuario el primer y último número a usar en lugar de 100 y 0 (pista: use el cuadro de diálogo `prompt`). Comprueba si los valores introducidos son correctos (que el valor inicial sea mayor que el valor final).

**Ejemplo**

```

let upperLimit = Number(prompt("Ingresa límite superior"));
let lowerLimit = Number(prompt("Ingresa límite inferior"));

if (!Number.isNaN(upperLimit) && !Number.isNaN(lowerLimit) && upperLimit > lowerLimit) {
    for (i = upperLimit; i >= lowerLimit; i -= 10) {
        console.log(i);
    }
}

```

app.js

Console>... 88  
78  
68  
58  
48  
38

**JS INSTITUTE**  
Open Education & Development Group

Ejemplo

**Tarea 2**

Modifica el programa anterior para que le pida al usuario el primer y último número a usar en lugar de 100 y 0 (pista: use el cuadro de diálogo `prompt`). Comprueba si los valores introducidos son correctos (que el valor inicial sea mayor que el valor final).

Ejemplo

**Tarea 3**

Hay diez números diferentes en este arreglo:

```
app.js
Console >...
88
78
68
58
```

Sandbox

edube.org dice  
Ingrresa limite superior  
55  
Aceptar Cancelar

```
    7
    8 }
```

Fullscreen

**JS INSTITUTE**  
Open Education & Development Group

Ejemplo

**Tarea 2**

Modifica el programa anterior para que le pida al usuario el primer y último número a usar en lugar de 100 y 0 (pista: use el cuadro de diálogo `prompt`). Comprueba si los valores introducidos son correctos (que el valor inicial sea mayor que el valor final).

Ejemplo

**Tarea 3**

Hay diez números diferentes en este arreglo:

```
app.js
Console >...
88
78
68
```

Sandbox

edube.org dice  
Ingrresa limite inferior  
10  
Aceptar Cancelar

```
    7
    8 }
```

Fullscreen

**JS INSTITUTE**  
Open Education & Development Group

Ejemplo

**Tarea 2**

Modifica el programa anterior para que le pida al usuario el primer y último número a usar en lugar de 100 y 0 (pista: use el cuadro de diálogo `prompt`). Comprueba si los valores introducidos son correctos (que el valor inicial sea mayor que el valor final).

Ejemplo

**Tarea 3**

Hay diez números diferentes en este arreglo:

```
let numbers = [21, 45, 100, 12, 11, 78, 61, 4, 39, 22];
```

Escribe un programa que primero escriba todos estos números en la consola, luego solo los que son pares (pista: el residuo de dividir un

```
app.js
Console >...
55
45
35
25
15
```

Sandbox

« 4.2.1.15 Bucles - Tareas »

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Tarea 3**

Hay diez números diferentes en este arreglo:

```
let numbers = [21, 45, 100, 12, 11, 78, 61, 4, 39, 22];
```

Escribe un programa que primero escriba todos estos números en la consola, luego solo los que son pares (pista: el residuo de dividir un número par entre 2 es igual a 0), luego solo los que son mayores que 10 y al mismo tiempo menor que 60.


**Tarea 4**

Escribe un programa utilizando un bucle que le pida al usuario el nombre de una película (primer cuadro de dialogo) y su calificación de [www.imdb.com](http://www.imdb.com) (segundo cuadro de dialogo). El programa te permitirá ingresar tantas películas como desees en el arreglo de películas. Cada elemento del arreglo será un objeto, que constará de dos campos: título e imdb. La entrada se completa si el usuario presiona Cancelar en el cuadro de diálogo. Luego, el programa debe imprimir primero en la consola todas las películas que tienen una calificación inferior a 7, luego aquellas cuya calificación sea mayor o igual a 7. Escribe el nombre de la película y su calificación uno al lado del otro, por ejemplo:</p>

Perdido en la Selva (7.7)


**Tarea 5**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

**Tarea 4**

Escribe un programa utilizando un bucle que le pida al usuario el nombre de una película (primer cuadro de dialogo) y su calificación de [www.imdb.com](http://www.imdb.com) (segundo cuadro de dialogo). El programa te permitirá ingresar tantas películas como desees en el arreglo de películas. Cada elemento del arreglo será un objeto, que constará de dos campos: título e imdb. La entrada se completa si el usuario presiona Cancelar en el cuadro de diálogo. Luego, el programa debe imprimir primero en la consola todas las películas que tienen una calificación inferior a 7, luego aquellas cuya calificación sea mayor o igual a 7. Escribe el nombre de la película y su calificación uno al lado del otro, por ejemplo:</p>

```
14 }
15
16 console.log("Películas con calificaciones inferiores a 7:");
17 for (movie of movies) {
18     if (movie.rating < 7) {
19         console.log(`${movie.title} (${movie.rating})`);
20     }
21 }
22
23 console.log("Películas con calificaciones superiores a 7:");
24 for (movie of movies) {
25     if (movie.rating >= 7) {
26         console.log(`${movie.title} (${movie.rating})`);
27     }
28 }
```

app.js

Películas con calificaciones inferiores a 7:  
 Películas con calificaciones superiores a 7:  
 Terror (10)


**Tarea 5**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

Ingrésa el título de la película

La roca

Aceptar Cancelar

```
20 }
21 ]
22
23 console.log("Películas con calificaciones superiores a 7:");
24 for (movie of movies) {
25     if (movie.rating >= 7) {
26         console.log(`${movie.title} (${movie.rating})`);
27     }
28 }
```

app.js

Películas con calificaciones inferiores a 7:  
 Películas con calificaciones superiores a 7:  
 Terror (10)


**Tarea 5**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

**Tarea 4**

Escribe un programa utilizando un bucle que le pida al usuario el nombre de una película (primer cuadro de dialogo) y su calificación de [www.imdb.com](http://www.imdb.com) (segundo cuadro de dialogo). El programa te permitirá ingresar tantas películas como desees en el arreglo de películas. Cada elemento del arreglo será un objeto, que constará de dos campos: título e imdb. La entrada se completa si el usuario presiona Cancelar en el cuadro de diálogo. Luego, el programa debe imprimir primero en la consola todas las películas que tienen una calificación inferior a 7, luego aquellas cuya calificación sea mayor o igual a 7. Escribe el nombre de la película y su calificación uno al lado del otro, por ejemplo:</p>

```
Perdido en la Selva (7.7)
```

**Ejemplo**

```
Console >_
Películas con calificaciones inferiores a 7:
Películas con calificaciones superiores a 7:
Terror (10)
```

**Tarea 5**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X



« 4.2.1.15 Bucles - Tareas »

**Tarea 4**

**Tarea 5**

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X



« 4.2.1.16 LABORATORIO: Bucles »

**LABORATORIO**

**Tiempo Estimado**

**Nivel de dificultad**

**Objetivos**

**Escenario**

```
10  "name: "Helen Richards",
11  "phone: "0800 1111",
12  "email: "libero@convallis.edu"
13 ];
14
15 // Función para buscar un contacto por nombre
16 function findContactByName(name) {
17   return contacts.find(contact => contact.name === name);
18 }
19
20 // Buscar el contacto por nombre
21 let resultado = findContactByName("Maxwell Wright");
22
23 // Imprimir el resultado en la consola
24 console.log(resultado);
25 |
```

app.js index.html style.css

Console >\_
[object Object]

## Modulo 5

## Sección 1



« 5.1.1.3 Funciones »

Sandbox

Echemos un vistazo a un programa simple, escrito sin ninguna función.

El programa calculará y mostrará la temperatura media diaria sobre la base de los datos proporcionada (24 mediciones de temperatura, en intervalos de una hora, a partir de la medianoche). En el programa declaramos las variable temperatures, que será una tabla de 24 elementos con las medidas obtenidas.

Tenemos medidas para dos días consecutivos, para lo cual haremos cálculos. La temperatura media, por supuesto, se calcula sumando todos los valores y dividiendo el resultado entre la cantidad.

A primera vista, puedes ver que el fragmento de código responsable de un cálculo se repite dos veces. En dos lugares del programa, usamos la misma secuencia de instrucciones, por lo que valdría la pena pensar en crear una función a partir de ellas.

Lo haremos en varias etapas, introduciendo algunos conceptos nuevos relacionados con las funciones. Comencemos con una declaración de función.

```
1 let temperatures;
2 let sum;
3 let meanTemp;
4
5 temperatures = [12, 12, 11, 11, 10, 9, 9, 10, 12, 13, 15, 18, 21, 24, 24, 23, 25, 25, 23,
6 sum = 0;
7 for (let i = 0; i < temperatures.length; i++) {
8 sum += temperatures[i];
9 }
10 meanTemp = sum / temperatures.length;
11 console.log(`mean: ${meanTemp}`); // -> media: 16.666666666666668
12
13 temperatures = [17, 16, 14, 12, 10, 10, 11, 13, 14, 15, 17, 22, 27, 29, 29, 27, 26, 2
14 sum = 0;
15 for (let i = 0; i < temperatures.length; i++) {
16 sum += temperatures[i];
17 }
18 app.js
```

Console>...  
mean: 16.666666666666668  
mean: 18.083333333333332

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X



« 5.1.1.4 Declaración de Funciones »

Sandbox

### Declaración de funciones

Al igual que con las variables, las funciones deben declararse antes de que podamos usarlas. La sintaxis para la declaración de funciones se ve así:

```
function functionName() {
    código
}
```

Este tipo de declaración de función en JavaScript se denomina **declaración de función**. Una instrucción de función comienza con la palabra clave `function` seguida del nombre de la función. Los nombres de las funciones deben seguir las mismas reglas que los nombres de las variables y también deben ser significativos. Después del nombre de la función, hay paréntesis que opcionalmente pueden tener parámetros de función, que discutiremos en un momento. Después de los paréntesis viene el cuerpo de la función, que se compone de cualquier cantidad de instrucciones (un bloque de código).

Intentemos declarar una función de acuerdo con estas reglas, que contendrá un fragmento de nuestro código de programa que calcula la temperatura media diaria. Lo llamaremos `getMeanTemp`. Por ahora, la función usará variables declaradas fuera de ella (en el contexto circundante). De hecho,

```
3 let meanTemp;
4
5 function getMeanTemp() {
6     sum = 0;
7     for (let i = 0; i < temperatures.length; i++) {
8         sum += temperatures[i];
9     }
10    meanTemp = sum / temperatures.length;
11
12 // Llamada a la función para calcular la temperatura media
13 getMeanTemp();
14
15 // Imprimir el resultado en la consola
16 console.log("La temperatura media es:", meanTemp);
17
18 app.js
```

Console>...  
La temperatura media es: 26.2

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X



« 5.1.1.6 Funciones - Variables Locales »

Sandbox

### Funciones - Variables Locales

Intentemos hacer un pequeño cambio en nuestro programa calculando la temperatura media. ¿Recuerdas qué son las variables locales? Así es como llamamos a las variables que se declaran y usan en un ámbito o alcance limitado y no son visibles en todo el programa, lo que significa que solo podemos usarlas dentro de ese ámbito en particular. Las variables declaradas con la palabra clave `let` son locales dentro del bloque de código (es decir, dentro del rango limitado por `{}`), mientras que las variables declaradas con la palabra clave `var` son locales dentro del bloque de funciones. Entonces, si declaras una variable dentro de un bloque de funciones, ya sea usando `let` o `var`, solo será visible (y utilizable) dentro del bloque de funciones. Esto es muy útil porque, por lo general, las variables que se usan dentro de una función no se necesitan fuera de ella.

En nuestro código, un ejemplo de tal variable es `sum`. Aunque la hemos declarado fuera de la función `getMeanTemp` (es una variable global), solo la referimos dentro de la función. Por lo tanto, una declaración global es innecesaria. Pongámoslo en orden, declarando `sum` localmente.

El comportamiento del programa es el mismo, pero el código ha ganado algo de claridad. La variable `sum` ahora es local, solo se puede usar dentro de la

```
1 let temperatures;
2 let meanTemp;
3
4 function getMeanTemp() {
5     let sum = 0;
6     for (let i = 0; i < temperatures.length; i++) {
7         sum += temperatures[i];
8     }
9     meanTemp = sum / temperatures.length;
10
11
12 temperatures = [12, 12, 11, 11, 10, 9, 9, 10, 12, 13, 15, 18, 21, 24, 24, 23, 25, 25, 23,
13 getMeanTemp();
14 console.log(`mean: ${meanTemp}`); // -> mean: 16.666666666666668
15
16 temperatures = [17, 16, 14, 12, 10, 10, 11, 13, 14, 15, 17, 22, 27, 29, 29, 27, 26, 2
17
18 app.js
```

Console>...  
mean: 16.666666666666668  
mean: 18.083333333333332

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

### La sentencia return

Las funciones que han sido invocadas ejecutan una secuencia de instrucciones contenidas en su cuerpo. El resultado de esta ejecución puede ser un valor determinado que tal vez queremos almacenar en alguna variable. La palabra clave `return` viene a ayudarnos en este caso. ¿Para qué sirve exactamente `return`?

- Primero, hace que la función termine exactamente donde aparece esta palabra, incluso si hay más instrucciones.
- Segundo, nos permite devolver un valor dado desde dentro de la función al lugar donde fue invocada.

Alejémonos por un momento de nuestro ejemplo con el cálculo de la temperatura media y juguemos con un código un poco más simple. La función `showMsg` contiene solo dos `console.logs` separados por `return`.

```
function showMsg() {
  console.log("mensaje 1");
  return;
  console.log("mensaje 2");
}
```

```
1 function showMsg() {
2   console.log("Hola");
3   return;
4   console.log("mensaje 2");
5 }
6 showMsg(); // -> mensaje|
```

app.js

Console &gt;\_

Hola

x Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## Modulo 5

### Sección 2

### Validación de parámetros

¿Recuerdas que dijimos que a veces usamos la palabra clave `return` para interrumpir funciones en caso de errores? Un buen ejemplo es la validación de parámetros de funciones.

Volvamos al ejemplo con la función `getMeanTemp`. La última versión que escribimos necesita un arreglo de números como argumento. Antes de comenzar el cálculo, podemos verificar si el valor que se le pasó es realmente un arreglo.

```
function getMeanTemp(temperatures) {
  if (!(temperatures instanceof Array)) {
    return NaN;
  }
  let sum = 0;
  for (let i = 0; i < temperatures.length; i++) {
    sum += temperatures[i];
  }
  return sum / temperatures.length;
}
```

```
1 function getMeanTemp(temperatures) {
2   if (!(temperatures instanceof Array)) {
3     return NaN;
4   }
5   let sum = 0;
6   for (let i = 0; i < temperatures.length; i++) {
7     sum += temperatures[i];
8   }
9   return sum / temperatures.length;
10 }
11
12 console.log(getMeanTemp(10));           // -> NaN
13 console.log(getMeanTemp([10, 30])); // -> 20
14 |
```

app.js

Console &gt;\_

NaN

20

x Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

## Expresiones de funciones

Para almacenar una función en una variable o pasárla como argumento para llamar a una función, no necesariamente tienes que declararla previamente y usar su nombre. Volvamos a nuestro ejemplo con la función add:

```
function add(a, b) {
    return a + b;
}

let myAdd = add;
console.log(myAdd(10, 20)); // -> 30
console.log(add(10, 20)); // -> 30
```

Primero declaramos la función add y luego la almacenamos en la variable myAdd. Podemos llamar a la misma función usando tanto el nombre add como la variable myAdd. Podemos acortar esta notación y declarar la función almacenándola en una variable.

```
let myAdd = function add(a, b) {
    return a + b;
}
```

```
1 let myAdd = function(a, b) {
2     return a + b;
3 }
4
5 console.log(myAdd(10, 20)); // -> 30
6
```

app.js

Console>\_

30

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

## Expresiones de funciones - continuación

Volvamos al concepto de funciones anónimas. Puede parecer un poco incomprendible cuando usas un nombre (aunque es un nombre de variable) para referirse a una función. En este caso, se trata de anónimo, es decir, la falta de un nombre, en la definición misma de una función. Esto será mucho más evidente al pasar una función como parámetro de llamada a otra función. Veamos el ejemplo:

```
function operation(func, first, second) {
    return func(first, second);
}

let myAdd = function(a, b) {
    return a + b;
}

console.log(operation(myAdd, 10, 20)); // -> 30

console.log(operation(function(a, b) {
    return a * b;
}, 10, 20)); // -> 200
```

```
1 function operation(func, first, second) {
2     return func(first, second);
3 }
4
5 let myAdd = function(a, b) {
6     return a + b;
7 }
8
9 console.log(operation(myAdd, 10, 20)); // -> 30
10
11 console.log(operation(function(a, b) {
12     return a * b;
13 }, 10, 20)); // -> 200
14
```

app.js

Console>\_

30

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

## Callbacks

Las funciones que se pasan como argumentos a otras funciones pueden parecer bastante exóticas y no muy útiles, pero de hecho, son una parte muy importante de la programación. Tan importante que incluso tienen su propio nombre. Son **funciones callback**. Como hemos visto en ejemplos anteriores, una función que recibe una callback como argumento puede llamarla en cualquier momento. Es importante destacar que, en nuestros ejemplos, la callback se ejecuta de forma síncrona, es decir, se ejecuta en un orden estrictamente definido que resulta de dónde se coloca entre las otras instrucciones.

### Callbacks síncronas

La ejecución **síncrona** es la forma más natural de ver cómo funciona el programa. Las instrucciones posteriores se ejecutan en el orden en que se colocan en el código. Si llama a una función, las instrucciones que contiene se ejecutarán en el momento de la llamada. Si le pasamos otra función a esta función como argumento, y también la llamamos dentro de una función externa, entonces todas las instrucciones mantendrán su orden natural.

```
let inner = function() {
```

```
1 let inner = function() {
2     console.log('inner 1');
3 }
4
5 let outer = function(callback) {
6     console.log('outer 1');
7     callback();
8     console.log('outer 2');
9 }
10
11 console.log('test 1');
12 outer(inner);
13 console.log('test 2');
14
```

app.js

Console>\_

test 1  
outer 1  
inner 1  
outer 2  
test 2

 Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

[Sandbox](#)

## Callbacks asíncronas

El funcionamiento **asíncrono** de programas es un tema bastante complejo, que depende en gran medida de un lenguaje de programación en particular y, a menudo, también del entorno.

En el caso de que JavaScript del lado del cliente se ejecute en un navegador, se limita a la programación basada en eventos, es decir, la respuesta asíncrona a ciertos eventos. Un evento puede ser una señal enviada por un temporizador, una acción del usuario (por ejemplo, presionar una tecla o hacer clic en un elemento de interfaz seleccionado) o información sobre la recepción de datos del servidor.

Usando las funciones apropiadas, combinamos un tipo específico de evento con una función callback seleccionada, que se llamará cuando ocurra el evento.

Uno de los casos más simples cuando existe una ejecución asíncrona de instrucciones es el uso de la función `setTimeout`. Esta función toma otra función (una callback) y el tiempo expresado en milisegundos como argumentos. La función callback se ejecuta después del tiempo especificado y, mientras tanto, se ejecutará la siguiente instrucción del programa (ubicada

```

1 let inner = function() {
2   console.log('inner 1');
3 }
4
5 let outer = function(callback) {
6   console.log('outer 1');
7   setTimeout(callback, 1000) /*ms*/;
8   console.log('outer 2');
9 }
10
11 console.log('test 1');
12 outer(inner);
13 console.log('test 2');

```

[app.js](#) [index.html](#) [style.css](#)
[Console](#) »...

test 1  
outer 1  
outer 2  
test 2  
inner 1

[Fullscreen](#)

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). [X](#)

[Sandbox](#)

## Las funciones `setTimeout` y `setInterval`

La función `setTimeout` se utiliza cuando deseas provocar una acción retrasada. Una función similar es `setInterval`. Esta vez, la acción también se realiza con retraso, pero periódicamente, por lo que se repite a intervalos fijos. Mientras tanto, se ejecuta el programa principal y, en el momento especificado, se llama a la función callback proporcionada como argumento para una llamada a `setInterval`.

Curiosamente, la función `setInterval` devuelve un identificador durante la llamada, que se puede usar para eliminar el temporizador utilizado en ella (y, en consecuencia, para detener la llamada cíclica de la función callback). Haremos esto en el siguiente ejemplo. Primero, ejecutamos `setInterval`, que llamará a la función callback (es decir, la función `inner`) en intervalos de un segundo. Luego llamamos a `setTimeout`, que apagará el temporizador asociado con el llamado previamente a `setInterval` después de 5.5 segundos. Como resultado, la función `inner` debe llamarse cinco veces. Mientras tanto, se ejecutará el resto del programa...

El resultado de la ejecución del programa debe ser los siguientes mensajes, que aparecerán en la consola:

```

1 let inner = function() {
2   console.log('inner 1');
3 }
4
5 let outer = function(callback) {
6   console.log('outer 1');
7   let timerId = setInterval(callback, 1000) /*ms*/;
8   console.log('outer 2');
9 }
10
11 setTimeout(function(){
12   clearInterval(timerId);
13 }, 5500);
14
15 console.log('test 1');
16 outer(inner);

```

[app.js](#) [index.html](#) [style.css](#)
[Console](#) »...

test 1  
outer 1  
outer 2  
test 2  
inner 1

[Fullscreen](#)

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). [X](#)

[Sandbox](#)

## Callbacks asíncronas - continuación

Si ejecutamos el código JavaScript en el lado del cliente, en el navegador, siempre está asociado con el sitio web. La ventana en la que se encuentra esta página se representa en el JavaScript del lado del cliente mediante una variable de ventana global. El objeto ventana tiene un método (o su propia función) llamado `addEventListener`. Esta función te permite registrar una determinada acción para que se realice en respuesta a un evento relacionado con la ventana. Dicho evento puede ser un "clic", que es un solo clic del mouse en cualquier lugar de la página (hay un conjunto limitado de eventos con nombre asociados con un objeto en particular, al que puede reaccionar). La acción a realizar se pasa al método `addEventListener` como una función callback.

```

1 // Agrega un event listener al objeto window para el evento de clic
2 window.addEventListener("click", function() {
3   console.log("Hizo clic en la ventana!");
4 });
5

```

[app.js](#)
[Console](#) »...

[Fullscreen](#)

## « 5.2.1.13 Funciones - Tareas »

**Tareas**
**Tareas 1**

Los arreglos en JavaScript tienen disponible un método `sort` que, como puedes suponer, te permite ordenar sus elementos. Este método toma como argumento una función que comparará los elementos del arreglo. La función debe devolver cero si consideramos que los argumentos son iguales, un valor menor que cero si consideramos que el primero es menor que el segundo y un valor mayor que cero en caso contrario. Mira el ejemplo:

```
let numbers = [50, 10, 40, 30, 20];
function compareNumbers(a, b) {
    let retVal = 0;
    if (a < b) {
        retVal = -1;
    } else if(a > b) {
        retVal = 1;
    }
    return retVal;
}
```

```
1 let numbers = [50, 10, 40, 30, 20];
2 function compareNumbers(a, b) {
3     let retVal = 0;
4     if (a < b) {
5         retVal = -1;
6     } else if(a > b) {
7         retVal = 1;
8     }
9     return retVal;
10 }
11 let sorted = numbers.sort(compareNumbers);
12 console.log(sorted); // [10, 20, 30, 40, 50]
```

**app.js**
**Console>...**

10,20,30,40,50

 Sandbox

 Fullscreen

## « 5.2.1.13 Funciones - Tareas »

**Tareas**
**Tareas 1**

Los arreglos en JavaScript tienen disponible un método `sort` que, como puedes suponer, te permite ordenar sus elementos. Este método toma como argumento una función que comparará los elementos del arreglo. La función debe devolver cero si consideramos que los argumentos son iguales, un valor menor que cero si consideramos que el primero es menor que el segundo y un valor mayor que cero en caso contrario. Mira el ejemplo:

```
let numbers = [50, 10, 40, 30, 20];
function compareNumbers(a, b) {
    let retVal = 0;
    if (a < b) {
        retVal = -1;
    } else if(a > b) {
        retVal = 1;
    }
    return retVal;
}
```

```
1 let numbers = [50, 10, 40, 30, 20];
2 let sorted = numbers.sort((a, b) => a - b);
3 console.log(sorted); // [10, 20, 30, 40, 50]
```

**app.js**
**Console>...**

10,20,30,40,50

 Sandbox

 Fullscreen

## « 5.2.1.13 Funciones - Tareas »

**Tarea 2**

Escribe tres funciones con los nombres `add`, `sub` y `mult`, que tomarán dos argumentos numéricos. Las funciones son para verificar si los argumentos dados son enteros (emplea `Number.isInteger`). Si no, devuelven `NaN`; de lo contrario, devuelven el resultado de la suma, la resta o la multiplicación, respectivamente. Las funciones deben declararse mediante una instrucción de función.

Ejemplo de su uso y resultados esperados:

```
console.log(add(12, 10)); // -> 22
console.log(sub(12, 10)); // -> NaN
```

**Ejemplo**

```
21
22 function div(a, b) {
23     if (!Number.isInteger(a) || !Number.isInteger(b)) {
24         return NaN;
25     }
26     if (b === 0) {
27         return "Error: División por cero";
28     }
29     return a / b;
30 }
31
32 console.log("Suma:", add(8, 3)); // Imprimirá: Suma: 8
33 console.log("Resta:", sub(10, 4)); // Imprimirá: Resta: 6
34 console.log("Multiplicación:", mult(6, 2)); // Imprimirá: Multiplicación: 12
35 console.log("División:", div(8, 2)); // Imprimirá: División: 4
36
```

**app.js**
**Console>...**

Suma: 8

Resta: 6

Multiplicación: 12

División: 4

 Sandbox

 Fullscreen

**Tarea 3**

Reescribe las funciones de la tarea anterior usando una expresión de función arrow o flecha, tratando de escribirlas en la forma más corta posible.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Tarea 4**

Escribe una función `action` que tomará la función callback como su primer argumento y los otros dos argumentos como números. Como función callback, podrá pasar una de las tres funciones de la tarea anterior. La función `action` llamará a la función callback que se le pasó y devolverá el resultado obtenido. La función callback aceptará el segundo y el tercer argumento de la invocación.

Ejemplo de su uso y resultados esperados:

```
console.log(action(add, 12, 10)); // -> 22
console.log(action(sub, 12, 10)); // -> 2
console.log(action(mult, 10, 10.1)); // -> NaN
```

**Ejemplo****Tarea 5**

Escribe un programa que imprima (en la consola) números enteros consecutivos 10 veces, en intervalos de dos segundos (comienza con el número 1). Utiliza las funciones `setInterval`, `clearInterval` y `setTimeout`.

Ejemplo de su uso y resultados esperados:

```
1
2
3
4
5
6
7
8
9
10
```

**Ejemplo**

```
5+ function sub(a, b) {
6  return a - b;
7 }
8
9+ function mult(a, b) {
10 return a * b;
11 }
12
13- function action(callback, num1, num2) {
14  return callback(num1, num2);
15 }
16
17 console.log(action(add, 12, 10)); // Imprimirá: 22
18 console.log(action(sub, 12, 10)); // Imprimirá: 2
19 console.log(action(mult, 10, 10.1)); // Imprimirá: NaN
20
```

app.js

```
Console>_
22
2
101
```

 Fullscreen**Tarea 5**

Escribe un programa que imprima (en la consola) números enteros consecutivos 10 veces, en intervalos de dos segundos (comienza con el número 1). Utiliza las funciones `setInterval`, `clearInterval` y `setTimeout`.

Ejemplo de su uso y resultados esperados:

```
1
2
3
4
5
6
7
8
9
10
```

```
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

app.js

```
Console>_
1
2
3
4
5
6
7
8
```

 Fullscreen**Tarea 6**

Escribe una función que calcule el  $n$ -ésimo elemento de la sucesión de Fibonacci. Esta secuencia se define mediante una fórmula:

$$\begin{array}{ll} 0 & \text{if } n = 0 \\ F_n = \begin{cases} 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases} \end{array}$$

Entonces, cada elemento de la secuencia (excepto los dos primeros) es la suma de los dos anteriores. Por ejemplo:  $F_1 = 1$ ,  $F_2 = F_1 + F_0 = 1$ ,  $F_3 = F_2 + F_1 = 2$  y  $F_4 = F_3 + F_2 = 3$ . La función debe usar **recursividad**. En la definición, usa una expresión de función (almacena una función anónima en una variable).

Ejemplo de su uso y resultados esperados:

```
console.log(fibbRec(4)); // -> 3
console.log(fibbRec(7)); // -> 13
```

```
1+ let fibbRec = function(n) {
2+   if (n <= 1) {
3+     return n;
4+   } else {
5+     return fibbRec(n - 1) + fibbRec(n - 2);
6+   }
7+ };
8
9  console.log(fibbRec(4)); // Imprimirá: 3
10 console.log(fibbRec(7)); // Imprimirá: 13
11
12
```

app.js

```
Console>_
3
13
```

 Fullscreen

 Sandbox

```
console.log(fibbRec(4)); // -> 3
console.log(fibbRec(7)); // -> 13
```

 Ejemplo

**Tarea 7**

Reescribe la función de la **Tarea 6** usando una expresión de función arrow o flecha, pero intenta acortar tu código tanto como sea posible (emplea operadores condicionales y trata de no usar variables adicionales que no sean el parámetro `n`).

```
1 let fibbRec = n => n <= 1 ? n : fibbRec(n - 1) + fibbRec(n - 2);
2
3 console.log(fibbRec(4)); // Imprimirá: 3
4 console.log(fibbRec(7)); // Imprimirá: 13
5
```

 Ejemplo

**Tarea 8**

Escribe una versión iterativa de la función de la **Tarea 6** (usa el bucle `for`). Declara la función usando una instrucción de función.

 app.js

 Console >\_

3

13

 Fullscreen

 Ejemplo

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

**Tarea 7**

Reescribe la función de la **Tarea 6** usando una expresión de función arrow o flecha, pero intenta acortar tu código tanto como sea posible (emplea operadores condicionales y trata de no usar variables adicionales que no sean el parámetro `n`).

```
1 function fibbIter(n) {
2   let fibPrev = 0;
3   let fibCurr = 1;
4
5   for (let i = 2; i <= n; i++) {
6     let temp = fibCurr;
7     fibCurr = fibPrev + fibCurr;
8     fibPrev = temp;
9   }
10
11   return fibCurr;
12 }
13
14 console.log(fibbIter(4)); // Imprimirá: 3
15 console.log(fibbIter(7)); // Imprimirá: 13
16
```

 app.js

 Console >\_

3

13

 Fullscreen

 Ejemplo

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

 Sandbox

**LABORATORIO**
**Tiempo Estimado**

30-45 minutos

**Nivel de Dificultad**

Medio

**Objetivos**

Familiarizar al estudiante con:

- Conceptos básicos de funciones (qué son las funciones, declaración de

```
1 let contacts = [
2   name: "Maxwell Wright",
3   phone: "(0191) 719 6495",
4   email: "Curabitur.@nonummyac.co.uk"
5 ], {
6   name: "Raja Villarreal",
7   phone: "0866 398 2895",
8   email: "posuere.vulputate@sed.com"
9 }, {
10  name: "Helen Richards",
11  phone: "00800 1111",
12  email: "libero@convallis.edu"
13 ];
```

 app.js

 index.html

 style.css

 Console >\_

 Fullscreen

**LABORATORIO**

### Tiempo Estimado

30-60 minutos

### Nivel de Dificultad

Medio

### Objetivos

Familiarizar al estudiante con:

- Conceptos básicos de funciones (qué son las funciones, declaración de funciones, llamada o invocación de funciones, variables locales, la sentencia return, los parámetros de las funciones, sombreado).
- Funciones como miembros de primera clase (expresiones de funciones, pasar una función como parámetro, funciones callback).
- Funciones arrow o flecha (declaración e invocación).
- Recursividad (concepto básico).

**Escenario**

```
17 contacts.sort((a, b) => {
18     if (a[field] > b[field]) return 1;
19     if (a[field] < b[field]) return -1;
20     return 0;
21 });
22
23 console.log("Contactos ordenados por", field + ":");
24 contacts.forEach(contact => {
25     console.log(contact.name, "=", contact[field]);
26 });
27 }
28
29 // Ejemplo de uso
30 let sortBy = "name"; // Puedes cambiar el campo por el que deseas ordenar (name, phone, etc)
31 sortContacts(sortBy);
32 
```

app.js index.html style.css

Console>...

Contactos ordenados por name:  
Helen Richards - Helen Richards  
Maxwell Wright - Maxwell Wright  
Raja Villarreal - Raja Villarreal

Fullscreen

## Modulo 6

### Sección 1

**¿Errores sin excepciones?**

En JavaScript, no todas las situaciones erróneas arrojan excepciones. Muchos de ellos se manejan de una manera ligeramente diferente. El mejor ejemplo son los *errores aritméticos*.

```
1 console.log(100 / 0); // -> Infinity
2 console.log(100 * "2"); // -> 200
3 console.log(100 * "abc"); // -> NaN
```

Ninguno de los comandos anteriores generará una excepción, aunque no parecen la aritmética más correcta. Dividir entre cero dará como resultado un valor `Infinity`. La multiplicación de un número por una cadena, que representaría un número, convertirá automáticamente esta cadena en un número (y luego realizará la multiplicación). Un intento de realizar una operación aritmética en una cadena que no representa un número (es decir, que no se puede convertir) dará como resultado `NaN` (no un número). Al menos dos de estos casos son claramente incorrectos (el primero y el tercero), pero en lugar de excepciones, la información sobre el error es el valor específico que se devuelve. Veamos un ejemplo más:

```
1 console.log(Math.pow("abc", "def")); // -> NaN
```

Console>...

Infinity  
200  
NaN

Fullscreen

## Confianza Limitada

Los programas no se ejecutan en el vacío. Por lo general, durante su ejecución, hay interacciones con los usuarios (p. ej., ingresar datos necesarios para calcular ciertos valores) u otros programas o sistemas (p. ej., descargar datos del servidor). El comportamiento tanto de los usuarios como de otros sistemas debe tratarse con precaución, y no podemos asumir que el usuario proporcionará datos en el formato que requerimos, o que el servidor de datos siempre funcionará. Tales situaciones inesperadas también serán fuentes de errores en nuestro programa. Y aunque no dependen directamente de nosotros, es nuestra responsabilidad anticiparnos a situaciones potencialmente peligrosas. Si, por ejemplo, escribimos una calculadora en la que el usuario ingresa sus valores, entonces probablemente deberíamos verificar si el divisor no es un cero antes de hacer la división. En teoría, el usuario debe saber que no dividimos entre cero, pero somos responsables de la estabilidad del programa. No le creas al usuario ni a otros sistemas. Debes suponer qué puede salir mal y verifica los datos recibidos antes de usarlos en tu programa.

Vamos a escribir un fragmento de código que te pedirá que introduzcas dos números. Luego queremos mostrar el resultado de dividir el primer número entre el segundo:

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

```
1 let sX = prompt("Ingrese el primer número");
2 let sY = prompt("Ingrese el segundo número");
3 let x = Number(sX);
4 let y = Number(sY);
5 if (Number.isFinite(x) && Number.isFinite(y) && y !== 0) {
6   console.log(x / y);
7 } else {
8   console.log("Argumentos incorrectos");
9 }
```

app.js

Console >...

2.75

 Fullscreen

22

Aceptar

Cancelar

 Sandbox

 Sandbox

## Confianza Limitada

Los programas no se ejecutan en el vacío. Por lo general, durante su ejecución, hay interacciones con los usuarios (p. ej., ingresar datos necesarios para calcular ciertos valores) u otros programas o sistemas (p. ej., descargar datos del servidor). El comportamiento tanto de los usuarios como de otros sistemas debe tratarse con precaución, y no podemos asumir que el usuario proporcionará datos en el formato que requerimos, o que el servidor de datos siempre funcionará. Tales situaciones inesperadas también serán fuentes de errores en nuestro programa. Y aunque no dependen directamente de nosotros, es nuestra responsabilidad anticiparnos a situaciones potencialmente peligrosas. Si, por ejemplo, escribimos una calculadora en la que el usuario ingresa sus valores, entonces probablemente deberíamos verificar si el divisor no es un cero antes de hacer la división. En teoría, el usuario debe saber que no dividimos entre cero, pero somos responsables de la estabilidad del programa. No le creas al usuario ni a otros sistemas. Debes suponer qué puede salir mal y verifica los datos recibidos antes de usarlos en tu programa.

Vamos a escribir un fragmento de código que te pedirá que introduzcas dos números. Luego queremos mostrar el resultado de dividir el primer número entre el segundo:

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

```
1 let sX = prompt("Ingrese el primer número");
2 let sY = prompt("Ingrese el segundo número");
3 let x = Number(sX);
4 let y = Number(sY);
5 if (Number.isFinite(x) && Number.isFinite(y) && y !== 0) {
6   console.log(x / y);
7 } else {
8   console.log("Argumentos incorrectos");
9 }
```

app.js

Console >...

2.75

 Fullscreen

88

Aceptar

Cancelar

 Sandbox

 Sandbox

## Confianza Limitada

Los programas no se ejecutan en el vacío. Por lo general, durante su ejecución, hay interacciones con los usuarios (p. ej., ingresar datos necesarios para calcular ciertos valores) u otros programas o sistemas (p. ej., descargar datos del servidor). El comportamiento tanto de los usuarios como de otros sistemas debe tratarse con precaución, y no podemos asumir que el usuario proporcionará datos en el formato que requerimos, o que el servidor de datos siempre funcionará. Tales situaciones inesperadas también serán fuentes de errores en nuestro programa. Y aunque no dependen directamente de nosotros, es nuestra responsabilidad anticiparnos a situaciones potencialmente peligrosas. Si, por ejemplo, escribimos una calculadora en la que el usuario ingresa sus valores, entonces probablemente deberíamos verificar si el divisor no es un cero antes de hacer la división. En teoría, el usuario debe saber que no dividimos entre cero, pero somos responsables de la estabilidad del programa. No le creas al usuario ni a otros sistemas. Debes suponer qué puede salir mal y verifica los datos recibidos antes de usarlos en tu programa.

Vamos a escribir un fragmento de código que te pedirá que introduzcas dos números. Luego queremos mostrar el resultado de dividir el primer número entre el segundo:

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

```
1 let sX = prompt("Ingrese el primer número");
2 let sY = prompt("Ingrese el segundo número");
3 let x = Number(sX);
4 let y = Number(sY);
5 if (Number.isFinite(x) && Number.isFinite(y) && y !== 0) {
6   console.log(x / y);
7 } else {
8   console.log("Argumentos incorrectos");
9 }
```

app.js

Console >...

2.75

 Fullscreen

## Confianza Limitada

Los programas no se ejecutan en el vacío. Por lo general, durante su ejecución, hay interacciones con los usuarios (p. ej., ingresar datos necesarios para calcular ciertos valores) u otros programas o sistemas (p. ej., descargar datos del servidor). El comportamiento tanto de los usuarios como de otros sistemas debe tratarse con precaución, y no podemos asumir que el usuario proporcionará datos en el formato que requerimos, o que el servidor de datos siempre funcionará. Tales situaciones inesperadas también serán fuentes de errores en nuestro programa. Y aunque no dependen directamente de nosotros, es nuestra responsabilidad anticiparnos a situaciones potencialmente peligrosas. Si, por ejemplo, escribimos una calculadora en la que el usuario ingresa sus valores, entonces probablemente deberíamos verificar si el divisor no es un cero antes de hacer la división. En teoría, el usuario debe saber que no dividimos entre cero, pero somos responsables de la estabilidad del programa. No le creas al usuario ni a otros sistemas. Debes suponer qué puede salir mal y verifica los datos recibidos antes de usarlos en tu programa.

Vamos a escribir un fragmento de código que te pedirá que introduzcas dos números. Luego queremos mostrar el resultado de dividir el primer número entre el segundo:

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

```

1 let sX = prompt("Ingrese el primer número");
2 let sY = prompt("Ingrese el segundo número");
3 let x = Number(sX);
4 let y = Number(sY);
5 if (Number.isFinite(x) && Number.isFinite(y) && y !== 0) {
6   console.log(x / y);
7 } else {
8   console.log("Argumentos incorrectos");
9 }
10

```

app.js

Console >...

 Fullscreen

0.25

## Modulo 6

### Sección 2

## Algunos detalles más sobre errores y excepciones de JavaScript

Tratemos de organizar la información sobre errores y excepciones, y sobre todo, su manejo. Esta vez, veamos el problema desde un punto de vista estrictamente funcional. Comenzaremos con una descripción general de los tipos de errores más importantes detectados por JavaScript, analizaremos con más detalle la sentencia `try...catch` y mostraremos que también podemos lanzar excepciones directamente.

### Tipos básicos de errores.

Existen algunos tipos de errores subyacentes que produce JavaScript. La mayoría de las veces, especialmente al principio, encontrarás errores de sintaxis y de referencia. También discutiremos los errores de tipo y rango.

#### 1. SyntaxError

Como dijimos anteriormente, un `SyntaxError` aparece cuando un código está mal formado, cuando hay errores tipográficos en las palabras clave, paréntesis o corchetes que no coinciden, etc. El código ni siquiera se puede ejecutar, ya que JavaScript no es capaz de entenderlo. Por lo tanto, se lanza el error correspondiente antes de que se inicie el programa.

```

1 "use strict";
2 iff (true) { //--> Uncaught SyntaxError: Unexpected token '('
3   console.log("true");
4 }
5

```

app.js

Console >...

 Fullscreen

Uncaught SyntaxError: Unexpected token '('

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). 

## 2. ReferenceError

Ya hemos visto este error. Ocurre cuando intentamos acceder a una función o variable que no existe. El motor de JavaScript no conoce el significado del nombre dado, por lo que es un error que calificaremos como un error semántico. La excepción correspondiente se lanza en el momento de la ejecución del programa, cuando se alcanza la instrucción incorrecta (es decir, en JavaScript, los errores semánticos son errores de tiempo de ejecución).

```
let a = b; // -> Uncaught ReferenceError: b is not defined
```

El intento de declarar la variable a no tiene éxito porque, al mismo tiempo, queremos inicializarla con el valor de la variable b. La variable b no se ha declarado en ninguna parte antes, por lo que el motor de JavaScript no conoce este nombre.

```
fun(); // -> Uncaught ReferenceError: fun is not defined
```

Esta vez, hemos intentado llamar la función fun. Si no la hemos declarado antes, y no hay ninguna función con este nombre entre las funciones estándar de JavaScript, la llamada finaliza con un error.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

Sandbox
Fullscreen

## 3. TypeError

Este tipo de error se produce cuando un determinado valor no es del tipo esperado (es decir, intenta realizar una operación que no es aceptable). Los ejemplos típicos son cambiar el valor constante o verificar la longitud de una variable que no es una cadena. Este error es particularmente importante cuando se trabaja con objetos, lo cual está fuera del alcance de este curso (hablaremos de ellos en la siguiente parte del curso). Este es un `run-time error` típico, por lo que se lanzará la excepción apropiada mientras se ejecuta el programa, después de llegar a la instrucción problemática.

```
const someConstValue = 5;
someConstValue = 7; // -> Uncaught TypeError: Assignment to constant variable.
```

Intentar almacenar el nuevo valor en la constante someConstValue falló por razones obvias, lo que resultó en un TypeError.

```
let someNumber = 10;
someNumber.length(); // -> Uncaught TypeError: someNumber.l
```

Esta vez, hemos intentado tratar el contenido de la variable `someNumber`.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

Sandbox
Fullscreen

## 4. RangeError

Este tipo de error se genera cuando pasas un valor a una función que está fuera de su rango aceptable.

Nuevamente, es un `run-time error`, y la excepción se lanza mientras el programa se está ejecutando, después de llegar a la instrucción incorrecta. De hecho, esta excepción es más útil al escribir tus propias funciones y manejar errores. Luego puedes lanzar una excepción en ciertas situaciones.

```
let testArray1 = Array(10);
console.log(testArray1.length); // -> 10
let testArray2 = Array(-1); // -> Uncaught RangeError: Invalid array length
console.log(testArray2.length);
```

En el ejemplo, hemos intentado crear dos arreglos usando el constructor (es decir, la función predeterminada) `Array`. Si pasamos un argumento a esta función, se tratará como el tamaño del arreglo recién creado. El primer arreglo (`testArray1`) se crea sin ningún problema. Como puedes adivinar, falla la creación del arreglo `testArray2` con una longitud negativa.

[F. Otras excepciones](#)

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

Sandbox
Fullscreen

**Sandbox**

## Manejo de excepciones condicionales

A veces queremos poder reaccionar de manera diferente a tipos específicos de errores dentro del bloque `catch`. Podemos hacer esto usando el operador `instanceof`. Hablaremos del operador en sí más adelante, porque es un tema bastante complicado. Por ahora, es suficiente saber cómo podemos usarlo para manejar errores. La sintaxis del operador `instanceof` tiene este aspecto:

```
variable instanceof type
```

Si, por ejemplo, recibimos un error en el bloque `catch` (pasado como argumento de error), podemos comprobar si es del tipo `ReferenceError` de la siguiente manera:

```
let result = error instanceof ReferenceError;
```

El operador `instanceof` devuelve un valor booleano, por lo que esta expresión devolverá `true` si la variable de error contiene un tipo `ReferenceError` y `false` si no es así. Podemos usar `if...else` o sentencias `switch` para luego ejecutar un código diferente en el caso de

The screenshot shows a browser-based JavaScript sandbox interface. At the top, there's a toolbar with icons for play, back, forward, download, and refresh, followed by a file menu and a "Sandbox" button. Below the toolbar is a code editor window containing the following code:

```
1 let a = -2;
2 try {
3   a = b;
4 } catch (error) {
5   if (error instanceof ReferenceError) {
6     console.log("Reference error, restablecer a a -2"); // -> Reference error, restabl
7     a = -2;
8   } else {
9     console.log("Otro error - " + error);
10 }
11 }
12 console.log(a); // -> -2
13 |
```

Below the code editor is a "Console" window showing the output of the script:

```
app.js
Console>...
Reference error, restablecer a a -2
-2
```

On the right side of the interface, there's a "Fullscreen" button.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**Sandbox**

## La sentencia finally

El último bloque opcional de la sentencia `try` es el bloque `finally`. Se puede usar con o sin el bloque `catch`, y siempre se ejecuta después de los bloques `try` y `catch`, independientemente de que se produzca algún error. La sintaxis para `try ... finally` se ve así:

```
try {
  // código a probar
} finally {
  // esto siempre se ejecutará
}
```

Hagamos un pequeño experimento. Haremos una sustitución correcta a la variable a dentro del bloque `try`:

```
let a = 10;
try {
  a = 5;
} finally {
  console.log(a); // -> 5
}
```

The screenshot shows a browser-based JavaScript sandbox interface. At the top, there's a toolbar with icons for play, back, forward, download, and refresh, followed by a file menu and a "Sandbox" button. Below the toolbar is a code editor window containing the following code:

```
1 let a = 10;
2 try {
3   a = b; // ReferenceError
4 } catch (error) {
5   console.log("Un error!"); // -> ¡Un error!
6 } finally {
7   console.log("Finalmente!"); // -> ¡Finalmente!
8 }
9 console.log(a); // -> 10
10 |
```

Below the code editor is a "Console" window showing the output of the script:

```
app.js
Console>...
¡Un error!
¡Finalmente!
10
```

On the right side of the interface, there's a "Fullscreen" button.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

**Sandbox**

## ¿Por qué deberíamos usar un bloque finally?

Esta es una buena pregunta, especialmente porque podemos lograr casi el mismo resultado simplemente escribiendo el código justo fuera de la instrucción `try...catch`, así:

```
let a = 10;
try {
  a = b; // ReferenceError
} catch (error) {
  console.log("Un error!");
}
console.log("Finalmente!";
```

Este código tendrá un resultado similar al del ejemplo anterior: registrará `¡Un error!` y luego `¡Finalmente!`. Es cierto que en este sencillo ejemplo, ambos scripts se comportarán igual, pero hay ligeras diferencias, y la más importante es que el bloque `finally` se ejecutará incluso cuando se arroje un error desde el bloque `catch`.

```
let a = 10;
try {
```

The screenshot shows a browser-based JavaScript sandbox interface. At the top, there's a toolbar with icons for play, back, forward, download, and refresh, followed by a file menu and a "Sandbox" button. Below the toolbar is a code editor window containing the following code:

```
1 let a = 10;
2 try [
3   a = b; // Primer ReferenceError
4 ] catch (error) {
5   try [
6     console.log(b); // Segundo ReferenceError
7   ] catch [
8     console.log("¡Segundo catch!"); // -> ¡Segundo catch!
9   ]
10 } finally {
11   console.log("¡Finalmente!"); // -> ¡Finalmente!
12 }
13 |
```

Below the code editor is a "Console" window showing the output of the script:

```
app.js
Console>...
¡Segundo catch!
¡Finalmente!
```

On the right side of the interface, there's a "Fullscreen" button.

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

☰ ⌂ ⚙
▶ ⏪ ⏴ ⏵ ⏷
Sandbox

## La instrucción throw y los errores personalizados

Existen varias razones por las que puedes generar tus propias excepciones. La mayoría de ellas son bastante complejas y no muy útiles en esta etapa de aprendizaje. La situación más fácil de imaginar es cuando escribes una función propia, que debería señalar los datos incorrectos que se le han pasado.

Para lanzar una excepción, usamos la instrucción `throw`. Le sigue cualquier valor que será tratado como una excepción. Puede ser, por ejemplo, un número o uno de los objetos de error preparados (por ejemplo, `RangeError`).

```
Una excepción que lancemos hará que el programa reaccione de la misma manera que las excepciones de JavaScript originales (es decir, detendrá su ejecución). Es decir, a menos que lo arrojemos dentro del bloque try para manejarlo. Veamos un ejemplo sencillo, sin tratar de encontrarle ningún significado especial. Esta es solo una ilustración del uso de la instrucción throw:
```

RangeError
Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

☰ ⌂ ⚙
▶ ⏪ ⏴ ⏵ ⏷
Sandbox

## Tareas

### Tarea 1

Escribe tu propia función `div` que tomará dos argumentos de llamada y devolverá el resultado de dividir el primer argumento entre el segundo. En JavaScript, el resultado de dividir entre cero es el valor `Infinity` (`-Infinity`, si intentamos dividir un número negativo). Cambia esto. Si se pasa `0` como segundo argumento, tu función deberá lanzar una excepción `RangeError` con el mensaje apropiado. Prepara una llamada de prueba de la función tanto para la división válida como para la división entre cero.

Ejemplo
Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).

☰ ⌂ ⚙
▶ ⏪ ⏴ ⏵ ⏷
Sandbox

### Tarea 2

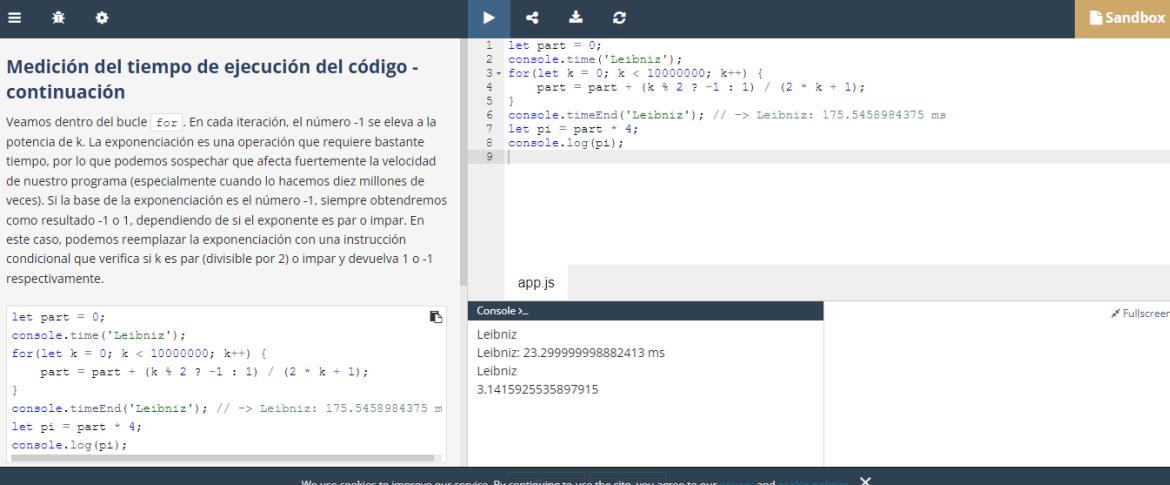
Hemos declarado un arreglo de números:

```
let numbers = [10, 40, 0, 20, 50];
```

Escribe un programa que, en un bucle, divida el número 1000 entre elementos sucesivos del arreglo de números, mostrando el resultado de cada división. Para dividir los números, usa la función de la tarea anterior. Usa la sentencia `try...catch` para manejar una excepción lanzada en el caso de la división entre cero. Si se detecta una excepción de este tipo, el programa debe imprimir un mensaje apropiado (tomado de la excepción) y continuar su operación (división por elementos sucesivos del arreglo).

Ejemplo
Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).



**Medición del tiempo de ejecución del código - continuación**

Veamos dentro del bucle `for`. En cada iteración, el número -1 se eleva a la potencia de k. La exponentiación es una operación que requiere bastante tiempo, por lo que podemos sospechar que afecta fuertemente la velocidad de nuestro programa (especialmente cuando lo hacemos diez millones de veces). Si la base de la exponentiación es el número -1, siempre obtendremos como resultado -1 o 1, dependiendo de si el exponente es par o impar. En este caso, podemos reemplazar la exponentiación con una instrucción condicional que verifica si k es par (divisible por 2) o impar y devuelva 1 o -1 respectivamente.

```

let part = 0;
console.time('Leibniz');
for(let k = 0; k < 10000000; k++) {
    part = part + (k % 2 ? -1 : 1) / (2 * k + 1);
}
console.timeEnd('Leibniz'); // -> Leibniz: 175.5458984375 ms
let pi = part * 4;
console.log(pi);

```

app.js

Console >...

Leibniz  
Leibniz: 23.29999999882413 ms  
Leibniz  
3.1415925535897915

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X



**Tareas**

**Tarea 1**

Ejecuta el siguiente código:

```

let end = 2;
for(let i=1; i<=end; i++) {
    console.log(i);
}

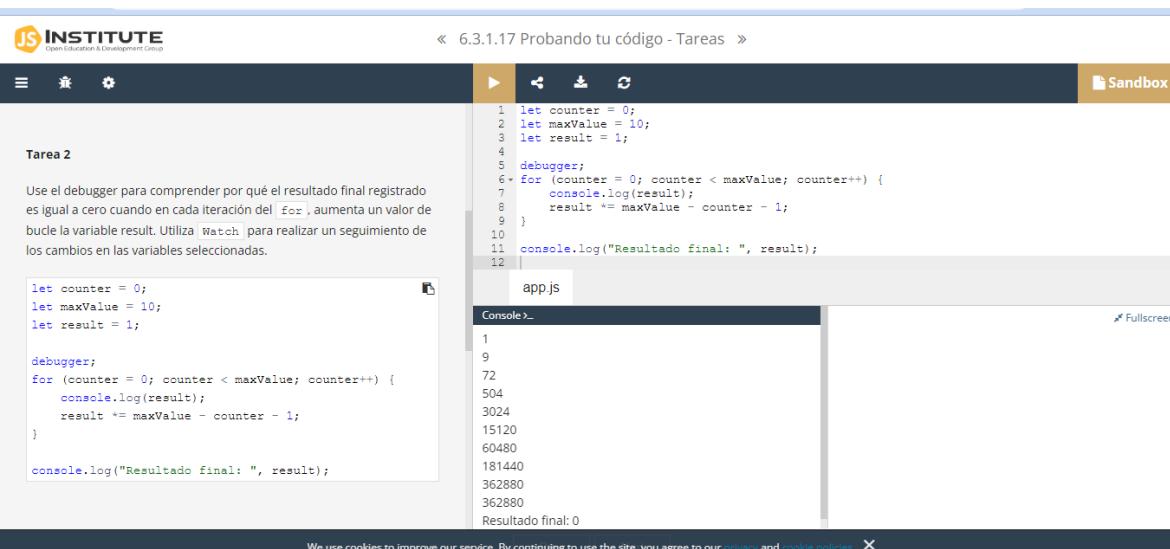
```

Debería dar como salida los números `1` y `2` en la consola. Usa el debugger para hacer que el programa genere los números `1`, `2`, `3`, `4` y `5`. No modifiques el código del programa. Usa solo puntos de interrupción y la opción de modificar variables.

**Tarea 2**

Use el debugger para comprender por qué el resultado final registrado

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X



**Tarea 2**

Use el debugger para comprender por qué el resultado final registrado es igual a cero cuando en cada iteración del `for`, aumenta un valor de bucle la variable `result`. Utiliza `Watch` para realizar un seguimiento de los cambios en las variables seleccionadas.

```

let counter = 0;
let maxValue = 10;
let result = 1;

debugger;
for (counter = 0; counter < maxValue; counter++) {
    console.log(result);
    result *= maxValue - counter - 1;
}

console.log("Resultado final: ", result);

```

app.js

Console >...

1  
9  
72  
504  
3024  
15120  
60480  
181440  
362880  
362880  
Resultado final: 0

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#). X

**Tarea 3**

Ejecuta el siguiente código:

```
function max(array) {
    let maxValue = array[0];
    for(let i=1; i<array.length; i++) {
        if(array[i] > maxValue) {
            maxValue = array[i];
        }
    }
    return maxValue;
}

console.log( max([1, 4, 6, 2])); // -> 6
console.log( max([10, 4, 6, 2])); // -> 6
```

La función max debe devolver el número más grande del arreglo dado como argumento. Como puedes ver, en el segundo caso funciona incorrectamente: en lugar del valor 10 obtenemos 6. Usando el depurador, rastrea la ejecución de la función max paso a paso. Observa los valores de las variables `i` y `maxValue`. Localiza el problema y

```
1 function max(array) {
2     let maxValue = array[0];
3     for(let i=1; i<array.length; i++) {
4         if(array[i] > maxValue) {
5             maxValue = array[i];
6         }
7     }
8     return maxValue;
9 }
10
11 console.log( max([1, 4, 6, 2])); // -> 6
12 console.log( max([10, 4, 6, 2])); // -> 6
13 |
```

app.js

```
Console>_
6
6
```

Fullscreen

We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).We use cookies to improve our service. By continuing to use the site, you agree to our [privacy](#) and [cookie policies](#).